MoDisco: a Model Driven Reverse Engineering Framework

Hugo Brunelière[a,1,*], Jordi Cabot[a], Grégoire Dupé[b], Frédéric Madiot[c]

[a] *AtlanMod (Inria & LINA), Ecole des Mines de Nantes, 4 rue Alfred Kastler, 44307 Nantes, France*
[b] *Mia-Software, 4 rue du Chateau de l'Eraudière, 44324 Nantes, France*
[c] *Obeo, 7 boulevard Ampère, Espace Performance La Fleuriaye, 44481 Carquefou, France*

Abstract

Context: Most companies, independently of their size and activity type, are facing the problem of managing, maintaining and/or replacing (part of) their existing software systems. These *legacy systems* are often large applications playing a critical role in the company's information system and with a non-negligible impact on its daily operations. Improving their comprehension (e.g., architecture, features, enforced rules, handled data) is a key point when dealing with their evolution/modernization.

Objective: The process of obtaining useful higher-level representations of (legacy) systems is called *reverse engineering* (RE), and remains a complex goal to achieve. So-called *Model Driven Reverse Engineering (MDRE)* has been proposed to enhance more traditional RE processes. However, generic and extensible MDRE solutions potentially addressing several kinds of scenarios relying on different legacy technologies are still missing or incomplete. This paper proposes to make a step in this direction.

Method: MDRE is the application of Model Driven Engineering (MDE) principles and techniques to RE in order to generate relevant model-based views on legacy systems, thus facilitating their understanding and manipulation. In this context, MDRE is practically used in order to 1) discover initial models from the legacy artifacts composing a given system and 2) understand (process) these models to generate relevant views (i.e., derived models) on this system.

Results: Capitalizing on the different MDRE practices and our previous experience (e.g., in real modernization projects), this paper introduces and details the MoDisco open source MDRE framework. It also presents the underlying MDRE global methodology and architecture accompanying this proposed tooling.

Conclusion: MoDisco is intended to make easier the design and building of model-based solutions dedicated to legacy systems RE. As an empirical evidence of its relevance and usability, we report on its successful application in real industrial projects and on the concrete experience we gained from that.

Keywords: Reverse Engineering, Legacy systems, Model Driven Engineering (MDE), Framework, Model Driven Reverse Engineering (MDRE)

*Corresponding author

Email addresses: hugo.bruneliere@inria.fr (Hugo Brunelière), jordi.cabot@inria.fr (Jordi Cabot), gdupe@mia-software.com (Grégoire Dupé), frederic.madiot@obeo.fr (Frédéric Madiot)

[1] Tel: +332 51 85 82 21

1. Introduction

Reverse engineering is almost as old as computer science itself. Initially targeting hardware analysis [1], it quickly extended its scope to mainly focus on software systems [2]. Then, following the spectacular expansion and advent of software from the end of the 80s, RE has been usually regarded in the context of dealing with *legacy systems* (i.e., already existing applications) which are often still running critical operations for companies.

In contrast with forward engineering, *reverse engineering* is commonly defined as *the process of examining an already implemented software system in order to represent it in a different form or formalism and at a higher abstraction level* [2]. The key notion is the one of *representation*, that can be associated to the concept of *model* in the large sense of the word.

The objective of such representations is to have a better *understanding* of the current state of a software system, for instance to correct it (e.g., to fix bugs or ensure regular maintenance), update it (e.g., to align it with the constantly evolving organizational policies and rules), upgrade it (e.g., to add new features or additional capabilities), reuse parts of it in other systems, or even completely re-engineer it. This is happening now more than ever, due to the need of not only satisfying new user requirements and expectations but also for adapting legacy systems to emerging business models, adhering to changing legislation, coping with technology innovation (in terms of used environments, frameworks, libraries, etc.) and preserving the system structure from deteriorating [3].

Clearly, given that reverse engineering is a time-consuming and error-prone process, any reverse engineering solution that could (semi)automate the process would bring precious help to the users (e.g. software architects or engineers) and thus facilitate its larger adoption. Nevertheless, such a solution would need to face several important problems:

- **Technical heterogeneity of the legacy systems**;
- **Structural complexity of these legacy systems**;
- **Scalability of the developed solution**;
- **Adaptability/portability of this solution**.

Previous attempts in the 90s to build (semi-)automated solutions were first based on object-oriented technologies [4]. Several proposals and tools in that area, e.g., aimed at *program comprehension* [5], appeared around at that time. Among the many proposals, some focused on the extraction and analysis of relevant information from existing source code or software components in general [6] whereas others focused on relational databases [7], compiled code or binary files [8], etc. However, these works were quite specific to a particular legacy technology or a given reverse engineering scenario (e.g., technical migration, software analysis).

With the emergence of Model Driven Engineering (MDE) [9], MDE principles and core techniques have been used in order to build effective reverse engineering solutions: this is the so-called **Model Driven Reverse Engineering (MDRE)** (cf. Section 2). MDRE formalizes the representations (models) derived from legacy systems to ensure a common understanding of their contents. These models are then used as the starting point of all the reverse engineering activities. Therefore, MDRE directly benefits from

the **genericity**, **extensibility**, **coverage**, **reusability**, **integration** and **automation** capabilities of MDE technologies to provide a good support for reverse engineering. Nevertheless, there is still a lack of complete solutions intended to cover full MDRE processes.

Based on previous works around MDRE plus our own concrete experience on software modernization, this paper presents MoDisco as 1) a generic, extensible and global MDRE approach and 2) a ready-to-use framework implemented as an official Eclipse project on top of the Eclipse platform. MoDisco has been created to address different types of legacy systems and to target different (model driven) reverse engineering processes such as technical/functional migration, refactoring, retro-documentation, quality assurance, etc. The paper collects and describes the research, experimentation and industrialization works performed around the MoDisco framework and overall MDRE approach during the last past years, largely extending the initial short descriptions of MoDisco as a project [10] [11].

The remainder of the paper is organized as follows. Section 2 introduces MDRE and summarizes its current main challenges and state of the art. Section 3 presents the MoDisco MDRE approach, which is actually implemented within the MoDisco project and corresponding MDRE framework (tool) as described in Section 4. Then, Section 5 illustrates the concrete use of MoDisco in two different industrial reverse engineering scenarios. Section 6 proposes a benchmark (and related results) to evaluate the performance of some MoDisco key components, while Section 7 gives general features and statistics about the MoDisco project and community. Finally, Section 8 compiles the main lessons learned from our global experience with MoDisco before Section 9 concludes the paper and opens with potential future works.

2. Model Driven Reverse Engineering: State of the Art & Challenges

Model Driven Reverse Engineering (MDRE) is commonly defined as the application of Model Driven Engineering (MDE) principles and techniques to the Reverse Engineering challenge.

2.1. Introducing Model Driven Engineering (MDE)

MDE states that many benefits can be gained by moving from usual code-centric approaches to more model-based or model-driven ones. This paradigm is largely based on the assumption that "Everything is a model" [12]. Thus, MDE basically relies on three main notions: *metamodel*, *model* and *model transformation*. A metamodel defines the possible element types and structure of the models which conform to it, similarly to the relationship between a grammar and corresponding programs in the programming field. Model transformations can be either model-to-model (e.g., Eclipse ATL [13]) or model-to-text (e.g., Eclipse Acceleo [14]) ones.

MDE is still currently a growing paradigm in Software Engineering. In this latest incarnation, it has been popularized by the Object Management Group (OMG) under the Model Driven Architecture (MDA) trademark [15] via modeling standards including a common metametamodel (Meta-Object Facility or MOF) [16], a general-purpose modeling language named Unified Modeling Language (UML) [17], transformation language specifications (Query/View/Transformation or QVT [18], MOF2Text [19]), etc. On the tooling side, Eclipse is the de-facto standard platform for MDE thanks to the Eclipse

Modeling Framework (EMF) [20] which notably provides a "reference" implementation of EMOF (Essential MOF, the set of MOF core concepts) named Ecore.

2.2. Model Driven Reverse Engineering (MDRE)

The application of MDE to Reverse Engineering (i.e., MDRE) is a relatively recent field [21]. At the beginning, models (in the MDE sense) were mainly used to specify systems prior to their implementation (forward engineering). Instead MDRE proposes to build and use models from the system implementation, thus directly benefiting from these higher-level views of the system (e.g., design models) or of its domain (e.g., application domain models) in order to improve the maintenance and evolution processes. MDRE is considered a fundamental application of MDE [22]. There is naturally an inherent complementarity between forward engineering and reverse engineering, especially when homogeneously treated and combined using models. This integration notably enables a continuous re-engineering of the systems.

The growing interest in MDRE motivated the OMG to launch the Architecture Driven Modernization (ADM) Task Force [23] with the main objective to propose a set of standard metamodels useful for modernization projects, i.e., technical migrations from old or obsolete technologies to more recent ones. Based on this, various proposals such as [24] combine these standards in a methodological framework. MDRE is also useful for software analysis purposes (e.g., as proposed by the Moose platform [25]). More generally, MDRE is required when dealing with *Model Driven Software Evolution* [26], i.e., global scenarios including any kind of possible modification on legacy systems (structural, functional, maintenance, etc.) and not only pure (technical) modernization. In all cases, a MDRE phase is needed first in order to obtain the required models from the considered systems so that they can then be analyzed, modified or evolved.

2.3. MDRE & RE: Current State of the Art

Language workbenches (also sometimes referred as metamodeling tools) have definitely paved the way for generic and extensible MDE frameworks like MoDisco by facilitating the creation of new metamodels like the ones used in MoDisco. Nevertheless, these initiatives (such as MetaEdit [27] and then MetaEdit+ [28]), GME [29] or MetaEnv [30] to mention just a few [2]) were more focused on the creation of a modeling environment for the new metamodel plus some support for typical (forward) engineering activities, i.e., code generation to some popular languages.

When it comes to tools / approaches focusing on reverse engineering, we first distinguish two main families: **specific** vs. **general purpose** solutions. This is determined depending on whether they aim to reverse engineer the system from a single technology and/or with a predefined scenario in mind (e.g., a concrete kind of analysis), or to be the basis for any other type of manipulation in later steps of the reverse engineering process. Of course, both lists are not completely disjoint and sometimes it can be argued that a tool could go either way. In the following, we review these two families and provide some representative examples for each one of them.

[2] A more complete list can be found in http://www.languageworkbenches.net

2.3.1. Specific Reverse Engineering Solutions

Several past works have already described how to migrate from a particular technology to another one using dedicated components and mappings, e.g., from procedural COBOL to object-oriented COBOL [31], from COBOL to Java [32], from COBOL to Java EE [33] or from Mainframe to Java EE [34]. These tools use specific and/or proprietary parsers, grammars, metamodels, etc. Contrary to the intent of MoDisco, their genericity and reusability in other contexts (e.g., other paradigms, legacy technologies or target environments) is quite limited. Moreover, MoDisco is clearly differentiating from them by being fully open source and relying on generic components.

MDE has also been applied in the context of legacy systems integration [35], including the reverse engineering of API concepts as high-level models. However, the main focus of this work was more on the mapping definition between different APIs and the generation of corresponding wrappers. MoDisco could obviously consider the obtained API models as useful inputs of more general MDRE processes but, as we have seen, it goes much more beyond this specific scenario.

Other particular examples of applications are automated design patterns detection using ontologies [36] or graphical interfaces separation and later reusability [37], which are also potentially adaptable in the context of MoDisco if some models can be exchanged between the solutions.

Regarding the model discovery phase, there exist several specific tools allowing to discover different types of models out of legacy artifacts. Columbus [38] is offering parsing capabilities from C/C++ source code and allows serializing the obtained information using different formats (e.g., XML, UML XMI). JaMoPP [39] or SpoonEMF [40] are providing alternative (but complete) Java metamodels and corresponding model discovery features from Java source code. Complementary to this, some other kinds of software artifacts can also be relevant as inputs of MDRE processes. For example, execution traces captured during the running of a given system have been used to generate UML2 sequence diagrams showing dynamic views on this system [41]. And in [42], available Web service descriptions have been expressed as UML models to be able to compose them for future integration in a different framework. All these components can be seen as potential *model discovers* that could be used either jointly with MoDisco or adapted to be plugged into the framework (cf. Section 4.5).

More related to software analysis, the ConQAT tool [43] is dedicated to the analysis of the source code (and also of related models, textual specifications, etc.) in order to perform quality assessment activities such as clone detection, architecture conformance or dashboard generation. As the MoDisco framework is not intended to address specifically software or quality analysis problems, it rather provides more generic components for helping people elaborating on solutions to various MDRE scenarios (potentially including software or quality analysis but not only).

Some other works are more focused on the related problem of software/program comprehension (or understanding). At code-level, GUPRO [44] is a solution, based on a central graph repository plus related queries and algorithms, that offers different visualizations over a given program (via tables, code excerpts, etc.). SWAG Kit [45] is another tool that is more specifically addressing the graphical visualization of complex C/C++ systems. Also, the CodeCrawler visualization tool [46] and more recently CodeCity [47], an advanced 3D graphical visualization tool (both based on the Moose platform, cf. sec-

tion 2.3.2), allow deeper analyzing the legacy code. The purpose of MoDisco is not to address the problem of advanced visualizations, but rather to provide relevant representations of legacy artifacts as models that could feed such visualization solutions if needed (cf. for instance the Sonar example in Section 5.2).

At artifact-level, [48] provides the capability of extracting key information from software artifacts (based on a generic artifact description metamodel) and supports traceability and consistency checking between them. MoDisco also offers representations of such artifacts and of (some of) their relationships as (KDM [23]) models that could complement the ones obtained from the application of this approach. The other way around, the information models produced by the proposed approach could be used as valuable inputs in MoDisco MDRE processes. But performing a further analysis of legacy artifact models for traceability or consistency purposes is not the current focus of MoDisco.

At the architecture level, a framework has been proposed [49] providing a formal definitions of architectural views (as diagrams/models) and an algorithm to perform consistency checks over these views. However there is no support for the automated discovery of these views, which is one of the main objectives of MoDisco. Similarly than before, MoDisco is not specifically intended to provide tooling for consistency checking issues (as this could be brought by other solutions such as this one).

2.3.2. Generic Reverse Engineering Platforms/Frameworks

Contrary to specific RE solutions/tools addressing particular RE activities or scenarios, there have been fewer initiatives to provide more generic integrated RE environments that could be extended and adapted to different application scenarios.

One of the first to appear was [50] that tried to integrate together various architecture and code re-engineering tools It is based on a common inter-operation schema called CORUM to be considered as a "standard" for the different tool providers in the area. The path MoDisco is following is relatively similar, i.e., using an horseshoe-like approach (that has been adapted to the MDE domain in our case). But MoDisco is not proposing to systematically conform to such a standard representation schema, it rather provides different base metamodels (e.g., from the OMG ADM Task Force[23] but not only) and also allows plugging other ones into the framework if required by a given MDRE scenario.

More recently, Moose[25] is a well-recognized platform providing a toolbox dedicated to the building of various software and data analysis solutions. Thus the Moose Suite is proposing a set of tools built-up around this same platform, all relying on a common fixed core named FAMIX that is dedicated to the representation of object-oriented systems. The main difference with MoDisco is that MoDisco is not relying on a single common core, but instead emphasizes a more open approach where technology-specific metamodels (for object-oriented technologies or others) are used to avoid information loss during the initial discovery phase. A more generic metamodel, e.g., to reuse already existing capabilities, could be used later on by transforming the specific models to such generic ones.

2.4. MDRE challenges

Taking into account this state-of-the-art and also our experiments of concretely applying MDRE in real projects (both from a research and industrial perspective), we believe MDRE has interesting capabilities to address the problems enumerated before (cf. Section 1) and thus could become more mainstream. More specifically, the main challenges MDRE solutions must be able to overcome are respectively:

- **To avoid information loss due to the heterogeneity of legacy systems.** To ensure the quality of the overall MDRE process, and thus validate its actual relevance, it is very important to be able to retrieve as much information as possible from legacy systems that are often technically heterogeneous.

- **To improve comprehension of the (typically complex) legacy systems.** The goal of MDRE is also to facilitate the understanding of structurally complex legacy systems by their users and developers. This requires going beyond the provision of simple low-level representations, and deriving efficiently higher abstract views with the most relevant information.

- **To manage scalability.** Legacy systems are usually huge and complex systems. Scalability of MDRE techniques must be improved to be able to load, query and transform in a suitable way the very large models usually involved in MDRE processes.

- **To adapt/port existing solutions to different needs.** Many MDRE solutions are still technology- or scenario- dependent, meaning that they target a very concrete legacy technology or reverse engineering scenario. Progress must be done in the development of generic MDRE solutions that, even if they still keep some specific components, are largely reusable in various contexts and for a minimized cost.

As a framework to facilitate the elaboration of MDRE solutions, MoDisco aims at providing support targeting the previous challenges. As stated before, MoDisco itself is not intended to be a complete solution for all specific scenarios (though its default components can be used as such for particular technologies), but its goal is rather to provide the basic (interconnected) blocks to build other MDRE solutions on top of it.

3. The MoDisco Model Driven Reverse Engineering (MDRE) Approach

This section presents the global MDRE approach we follow in MoDisco to treat as homogeneously as possible all potential reverse engineering scenarios. In order to try to overcome the MDRE challenges pointed out before, our objective is 1) to identify the main steps and components commonly used in MDRE solutions and 2) to assemble them coherently as a generic and extensible approach.

3.1. Overall approach

We believe a full MDRE approach must provide the following characteristics in order to fulfill the expressed requirements:

- **Genericity** based on technology-independent standards (i.e., metamodels) and customizable model-based components where specific technology support is ensured by additional dedicated features that can be plugged into the generic ones;

- **Extensibility** relying on a clear decoupling of both the represented information (as models) and the different consecutive steps of the process (as MDE operation workflows);

- **Full coverage** (if necessary) of the source artefacts based on complementary interrelated views of the same system at different abstraction levels and using different viewpoints (i.e., metamodels);
- Direct **(re)use** and **integration** of both the provided components and possibly external ones, but also of all the obtained results (i.e., models);
- Facilitated **automation** of (at least parts of) the process thanks to the already available MDE techniques, notably by chaining predefined sets of model transformations.

To achieve them in MoDisco, we follow the strategy of switching very early from the heterogeneous world of legacy systems to the more homogeneous world of models. Thus, we can directly benefit as soon as possible from the interesting properties of MDE and the full set of already available technologies (e.g., for manipulating the models of the legacy systems).

The main principle is to quickly get *initial models* representing the artifacts of the legacy system without losing any of the information required for the process. These "raw" models are sufficiently accurate to be used as a starting point of the considered MDRE scenario, but do not represent any real increase of the abstraction or detail levels. They can be seen as (full or partial) abstract syntax graphs that bridge the syntactic gap between the worlds (i.e., technical spaces) of source code, configuration files, etc. and the world of models. From this point on, any reverse engineering task to be performed on the system can be done with the same result using these models as a valid input representation. Therefore, we have reduced the heterogeneity of the reverse engineering process to a modeling problem.

The models can then be used as inputs for chains of MDE operations, e.g., model transformations, in order to navigate, identify and extract the relevant information. These more "processed" models obtained at the end of the chain, called *derived models*, are then finally used to generate and/or display the expected views of the reverse engineered legacy system.

As presented in Section 2, MDRE is the application of MDE in a Reverse Engineering context. In order to realize it, the two main steps mentioned right before are summarized by this simple equation:

Model Driven Reverse Engineering

=

Model Discovery + Model Understanding

Model Discovery is a **metamodel-driven** phase in charge of representing the legacy system as a set of models with no loss of required information (Section 3.2). The **Model Understanding** phase is completely **model-based**: it takes as input the models from the previous phase and generates the required output (models) thanks to a chain of model transformations (Section 3.3). While the components to use in the model discovery phase largely depend on the source technology to analyze (i.e., the metamodels to employ are for instance adapted to the legacy system programming languages), the components in the model understanding phase are more related to the actual objective

of the overall reverse engineering process itself. Such a goal could be simply system comprehension (e.g. of its architecture or implemented functionalities), further analysis (e.g. business rule extraction, non-functional property verification), re-engineering (e.g. technical refactoring or migration), etc.

To structure these phases and components in practice, we propose the global architecture depicted in Figure 1 (cf. Figure 6 for a concrete instantiation), which is composed of three vertical complementary layers that favor the five main characteristics described before:

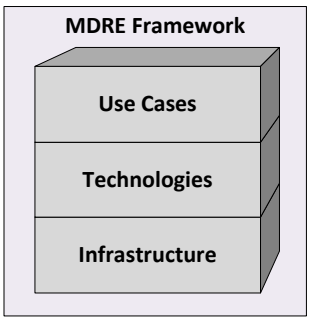

Figure 1: MDRE framework architecture

The *Infrastructure* layer provides genericity and automation via a set of basic bricks totally independent from any legacy technology and reverse engineering scenario. Such components offer for instance generic metamodels and model transformations. They also propose extensible model navigation, model customization and model orchestration capabilities (manual via dedicated user interfaces and/or programmatic via specific APIs). They often come with the corresponding generic interfaces and extension features required for the components from the other layers to be plugged in.

The *Technologies* layer is built on top of the *Infrastructure* one. It offers (partial or full) coverage for some legacy technologies and also gives the chance to extend this coverage to other ones. Its goal is to provide technology-dedicated components which however stay independent from any specific reverse engineering scenario. Such components can be either technology-specific metamodels or their corresponding model discoverers (cf. Subsection 3.2), as well as related transformations (cf. Subsection 3.3). They are the concrete bricks addressing the different (kinds of) legacy systems to be potentially reverse engineered.

Finally the *Use Cases* layer provides some reuse and integration examples, which are either relatively simple demonstrators or more complete ready-to-use components implementing a given reverse engineering process. Such components are mostly intended to realize the actual integration between components from the two other layers, and can be either reuse as-is or extended/customized for a different scenario.

According to these principles and architecture, the next subsections details the two main MDRE phases namely *Model Discovery* and *Model Understanding*.

3.2. Model Discovery

Model Discovery is the action of automatically obtaining raw model(s) from the legacy system to be reverse engineered [3]. These models are *initial models* because they have a direct correspondence with the elements of the legacy system: they can be considered as (full or partial) abstract syntax graphs as they do not imply any deeper analysis, interpretation or computation at this stage. To be able to represent the legacy system without losing any information, the metamodels employed at this stage are often metamodels of a low abstraction level and that closely resemble the source technology such as a general programming language (e.g., Java, CSharp, C++), a Domain Specific Language or DSL (e.g., SQL, HTML), a file format (e.g., Microsoft Word or Excel), etc.

The metamodel-driven software components allowing to generate these raw models are called *model discoverers*, where each discoverer targets a specific (legacy) technology. The overall principle of a model discoverer is shown on Figure 2.

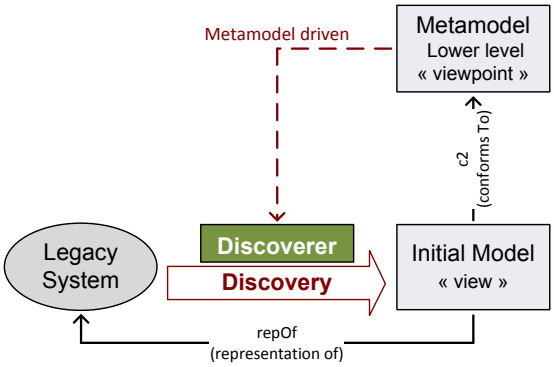

Figure 2: General principle of Model Discovery

First the metamodel (defining the viewpoint the system will be observed from) has to be designed, depending on the kind of legacy system that is concerned. For some well-known technologies like Java/JEE, appropriate metamodels can already be available (e.g., see the ones included in MoDisco as presented in the next section). This metamodel is the main factor ensuring the quality and completeness of the discovery phase in charge of creating instances of this metamodel from the legacy system. The model resulting from the execution of the built discoverer will conform to this base metamodel.

In some cases, the discovery process can be split up into two complementary steps:

1. **Injection** - focusing on bridging between the technical spaces of the legacy system and the modelware technical space of MDE, e.g., using parsers or APIs to access the content of legacy artifact(s) and then create the model accordingly;
2. **Transformation** - an additional syntactic (or structural) mapping inside the MDE technical space to produce the actual initial model from the result (i.e. a model) of the previous injection step, if required.

[3] Even if some models were created during the development process, they could already be outdated compared to its actual state. In that case, model discovery can provide a useful complementary support to validate the relevance of these models.

These two steps are not necessarily atomic operations performed at once and can rather be composed of several sub-operations possibly iterative and interactive, notably chains of model transformations. The length of this chain depends on the complexity of the discovery process and on the availability of some intermediate metamodels that can be used to simplify it.

For instance, considering the case of discovering the model corresponding to a set of source code files in Java, the discovery process implementation can vary:

- The **two-step approach** can be strictly applied: a generic (intermediate) model of the abstract syntax tree can be obtained first from the program (based on a generic metamodel such as OMG ASTM [23], e.g., as in [51]) and then transformed into a language-specific (low level) model (i.e., a Java model);
- A more **direct approach** can be considered: visit the abstract syntax tree of the program (e.g., using a dedicated API or the result of a previous parsing) in order to build directly the language-specific (low level) model (i.e., the Java model), the *transformation* step being integrated in the *injection* one.

There are different arguments to help deciding between these two implementation options. In the first case, considering such a two-step process usually allows reducing the complexity of each individual step and thus can facilitate the global discoverer implementation. The first part of the discoverer (the injector) becomes generic and can be directly reused by similar model discoverers. Nevertheless, when these benefits are not so relevant for the reverse engineering scenarios interesting for the user, one can decide to go for the second option which involves less artifacts to think about and offers better performance. Anyway, the *initial models* obtained at the end of both alternatives represent the same system at the same level of details.

3.3. Model Understanding

The *initial models* discovered in the previous phase can be exploited in different ways in order to obtain the final expected representations of the reverse engineered legacy system. Thus the Model Understanding phase as described in Figure 3 is mainly transformation-based: it largely relies on (chains of) model transformations to perform semantic mappings (in the data sense) that generate a set of *derived models*, according to the information/structure expected by the reverse engineer from the initial models obtained in the previous step. The outputs can be the *derived models* themselves or the result of extracting these models into some external tools (e.g., for visualization purposes).

Several *derived models*, temporary or final, can be obtained from the same *initial models* using different automated (chains of) model-to-model transformations depending on the goal of the reverse engineering process. This is an important benefit of such a two-phase approach, as reusing the discovery phase facilitates a lot the exploitation and analysis of the legacy system. Each one of these derived models (views) conforms to a metamodel (viewpoint) of, usually, a higher abstraction level [4] and that is tailored to the required view of the system or to the targeted external tool.

[4] An exception would be a re-engineering scenario where the goal is to re-implement the system using a new technology. In this case, lower level models (representing the system in this new technology) are required for a final model-to-text transformation to actually generate the corresponding source code

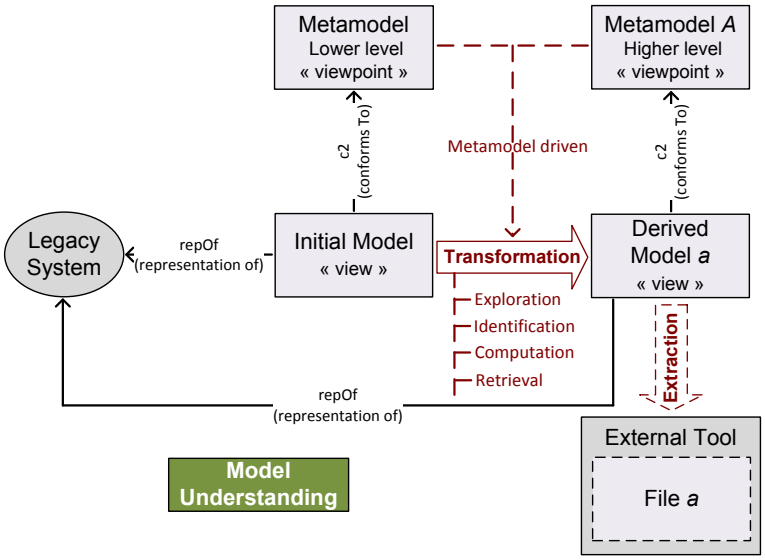

Figure 3: General principle of Model Understanding

As shown on Figure 3, a Model Understanding process generally includes the following main actions, all performed by/within **model transformations** and usually executed in an iterative process that refines the results until the desired derived models are obtained:

1. Legacy system **exploration** via its *initial models* (**model navigation**);
2. Required information **identification** via these models (**model querying**);
3. View **computation** using the identified information as source (**model computation**);
4. Representation **retrieval** into *derived models* (**model building**).

The legacy system exploration is performed on the *initial models* of that system rather than directly on it. Thus, extended model navigation capabilities (cf. the ones provided in model transformation via dedicated languages such as OCL [52] for instance), are required to browse them efficiently. This implies notably navigating inside these models at all detail levels and so returning as a result different sets of model elements, structural features (i.e., attributes and references), annotations, etc. These results are then queried, often iteratively, in order to select/filter and obtain only the information strictly needed. This information is used in order to perform the actual computation of the expected views over the system represented by the initial models.

As said before, all these steps can be implemented by model transformations specifying the corresponding refinements. Such transformations can be defined using different languages that can be declarative (e.g., QVT [18]), imperative (e.g., Java, Kermeta [53]) or hybrid (e.g., ATL [54]). Basically, they take as input(s) the navigated/queried model(s) and generate as output(s) the computed model(s) that can conform to the same metamodel, its augmented or reduced version, or a totally different one.

The target models (or representations) are built as the final results from the various obtained *derived models* (i.e., refinements). Thanks again to other model transformations

(e.g. targeting visualization formats such as SVG or DOT) but also sometimes to external tools (e.g. being able to render models graphically), these last refinements are actually used to retrieve the expected representations of the initial legacy system.

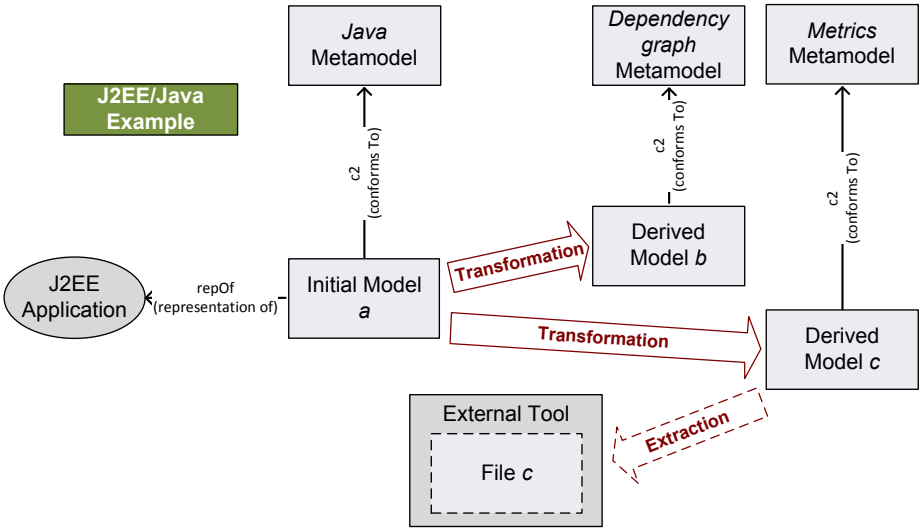

Figure 4: J2EE/Java Example - Model Understanding

As an example, Figure 4 shows a possible reverse engineering scenario from Java source code. The *initial model* is the complete Java model previously discovered from a given Java EE application. This model can be then used to generate different views of the system, like the dependency graph of the Java classes according to the internal method calls or a set of metrics on the complexity of the code such as the number of classes per package, the average number of methods per class, etc. These metrics can be finally sent to an external reporting tool able to provide graphical analytical representations for the provided data.

4. The MoDisco MDRE Framework

This section describes the MoDisco framework implementing the generic, extensible and customizable MDRE approach described in the previous section. The goal of MoDisco is to facilitate the development of model-based and model-driven solutions targeting different reverse engineering scenarios and legacy technologies.

Next subsections present the MoDisco project in detail, as well as the MoDisco technology and infrastructure layers supporting both the model discovery and model understanding phases.

4.1. MoDisco Project Overview

MoDisco is an open source project officially part of the Eclipse Foundation [55] and is integrated in the Modeling top-level project promoting MDE techniques and their development within the Eclipse community. It is also today officially recognized and

quoted by the OMG as providing the reference implementations and tooling of several of the ADM task force industry standards [23]:

- The Knowledge Discovery Metamodel (KDM);
- The Software Measurement Metamodel (SMM);
- The Generic Abstract Syntax Tree Metamodel (GASTM).

Initially created as an experimental research framework by the AtlanMod Team [56], the project has now evolved to an industrialized solution thanks to an active collaboration with the MIA-Software company [57].

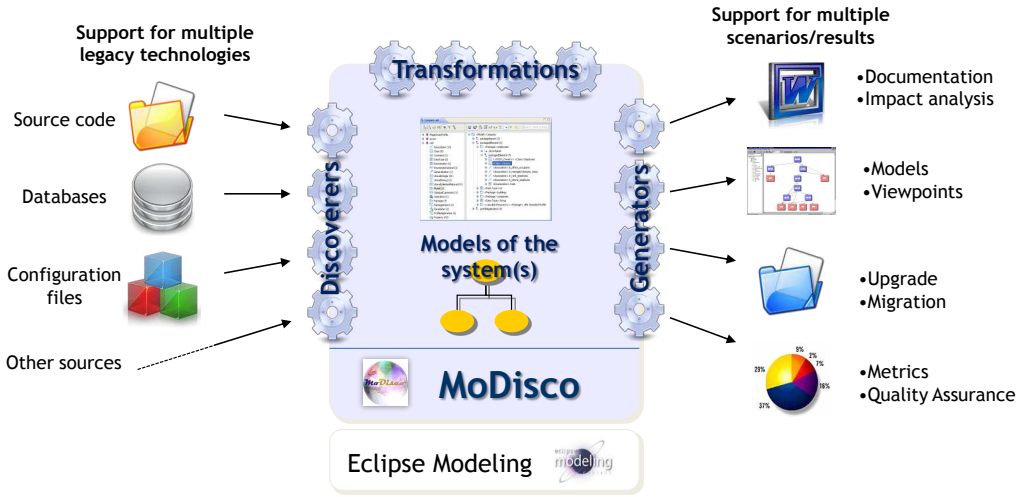

Figure 5: Overview of the Eclipse-MDT MoDisco project

As shown on Figure 5, from all kinds of possible legacy artifacts (e.g. source code, databases, configuration files, documentation, etc.) MoDisco aims at providing the required capabilities for creating the corresponding model representations and allowing their handling, analysis and use. Relying on the Eclipse Modeling Project and notably the Eclipse Modeling Framework (EMF) [20], it offers various components such as discoverers, transformations, generators, etc. to implement this support (cf. the next subsections for more details). As output, the framework targets the production of different views/artifacts on/from the considered legacy systems, depending on the expected usage of the reverse engineering results (e.g., software modernization, refactoring, retro-documentation, quality analysis, etc.). One of the main goals of MoDisco is to remain adaptable to many different scenarios, thus facilitating its adoption by a potentially larger user base.

The MoDisco framework has been integrated in the latest Eclipse Simultaneous Releases, namely Helios, Indigo, Juno and Kepler. These releases provide several ready-to-use Eclipse bundles targeting different families of user, and notably the MDE engineers via the *Eclipse Modeling Tools* bundle MoDisco is actually part of. This year again,

MoDisco is going to be part of the coming Eclipse Luna Simultaneous Release (June 2014) along with the other main Eclipse Modeling projects.

MoDisco is structurally delivered as a set of Eclipse features and related plug-ins whose builds are directly downloadable via the MoDisco update sites. Each MoDisco component is actually composed of one or more plug-ins and can be classified according to the previously described three-layer architecture, as summarized on Figure 6 and detailed in the next subsections.

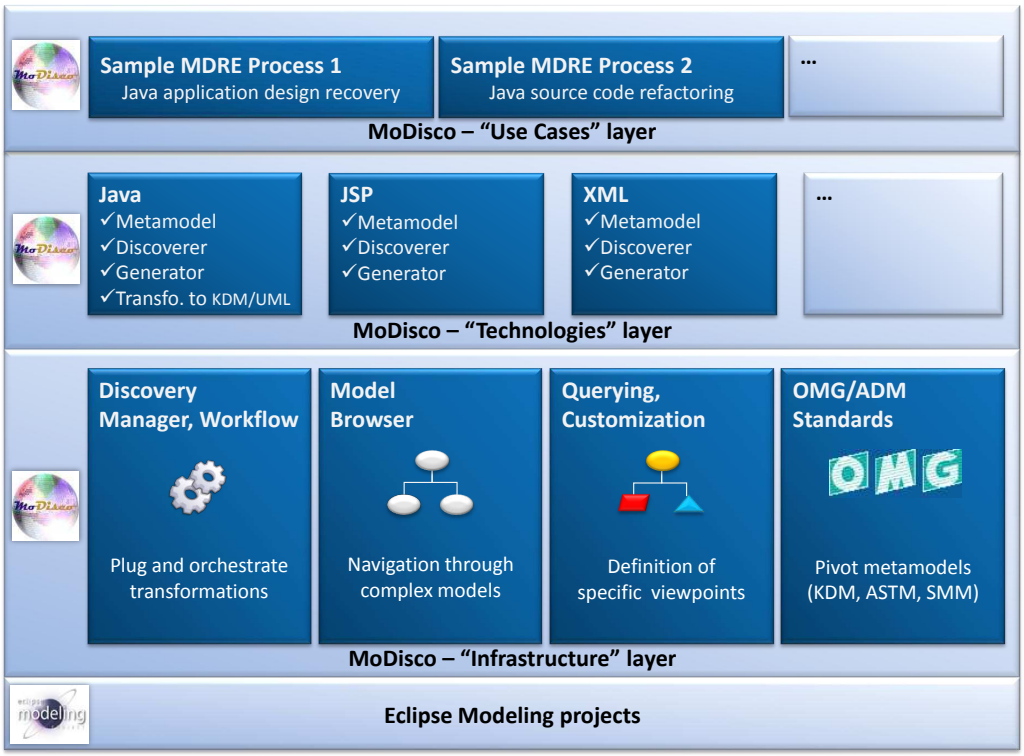

Figure 6: Overview of the MoDisco framework's architecture

Together with generic components (the infrastructure) allowing to create dedicated MDRE solutions, MoDisco also provides predefined technology-specific components that allow users to directly target some types of legacy artifacts and use cases. It is worth to note that some MoDisco components have been deemed useful beyond a strictly reverse engineering context and are now being externalized to facilitate their reuse in other projects, e.g., EMF Facet [58] as explained later in subsection 4.2.

Apart from these components, the MoDisco project is also equipped with all the standard Eclipse tooling to support its development and the relation with its user community (cf. Section 7).

4.2. MoDisco Infrastructure Layer

As part of its infrastructure layer, MoDisco currently provides a set of generic components that are relevant independently from the concerned legacy technologies or reverse

engineering scenarios.

4.2.1. OMG/ADM Standards

From a standardization point of view, concrete implementations of three OMG ADM standard metamodels [23] are available, namely KDM, SMM and GASTM as introduced before. KDM allows representing the entire software system and all its entities at both structural and behavioral levels. It deals with the legacy system metadata, artifacts and higher level structure of the code in a generic way, while GASTM focuses more on the lower level abstract syntax tree of the sources (independently from the used language). SMM is used to both specify any kind of measure/metric on legacy software and express the obtained results (measurements). All of them come with an EMF Ecore version of the metamodel in addition to the generated model handling API. Moreover, there is a more advanced support for KDM. A corresponding model discoverer (using a model transformation to KDM as explained in subsection 4.3) allows the automated analysis and representation of the file hierarchy of applications as so-called KDM "Source" models (using a subset of KDM), and a predefined transformation allows obtaining UML2 models (class diagrams in that case) from KDM "Code" models (using another subset of KDM). Also, a small framework has been developed based on the KDM metamodel to facilitate the future building of new model discoverers mixing both physical resource metadata (KDM models) and code content information (e.g., Java models).

4.2.2. Discovery Manager and Workflow

To globally manage all the model discoverers registered within the MoDisco environment, a Discovery Manager is provided. It comes along with the simple generic interface a discoverer should implement in order to be plugged into the framework (plus the corresponding extension point), and also with a Discoverers View providing a quick view over all registered discoverers. In addition, the MoDisco Workflow enables the chaining and launching of a set of registered discoverers, transformations, scripts, etc. as part of larger MDRE processes (cf. Subsection 4.4).

4.2.3. Model Browser and Navigation

One of the most powerful MoDisco component is the Model Browser. It has been designed to make the navigation through complex models much easier by providing advanced features such as full exploration of all model elements, infinite navigation (as being a graph, the model is fully navigable and not restricted by a tree representation), filtering, sorting, searching, etc. It is mainly composed of two panels: the left panel is displaying the possible element types (i.e., the concepts from the concerned metamodel) while the right panel is showing the model elements themselves. It is also completely metamodel-independent and customizable (via the definition of specific customization models). For instance the icons and global formatting of the displayed information can be specialized for a given metamodel, as shown with UML2 in Figure 7.

4.2.4. Querying and Customization

MoDisco is also equipped with model querying capabilities that are particularly relevant when elaborating on MDRE processes. Thus, the component named Query Manager allows registering, gathering and executing model queries over any kind of model. A dedicated metamodel has been designed, along with the corresponding tooling, for the users

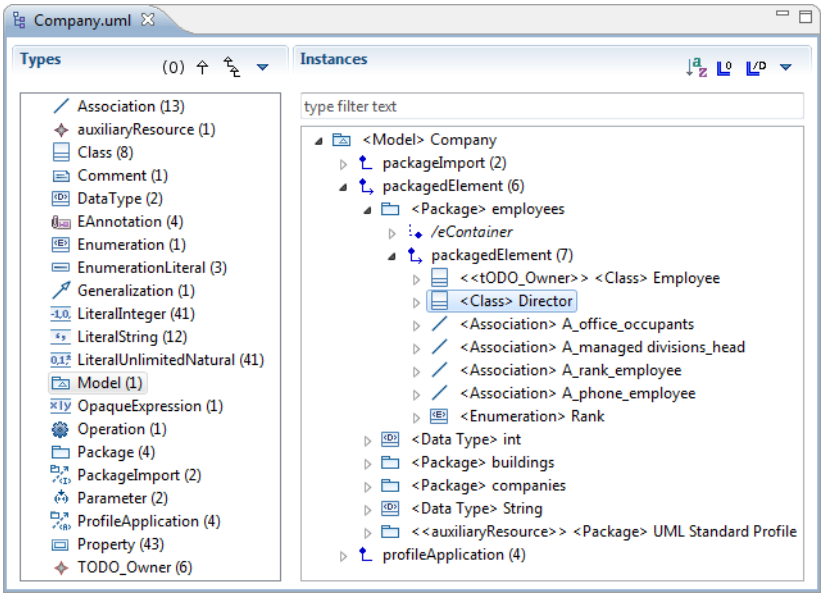

Figure 7: Generic MoDisco Model Browser, customized for the UML2 metamodel

to be able to describe their query sets as models. The key point is that the offered mechanism is fully query language-independent: queries can currently be written in Java, OCL, JXPath, and new drivers for other languages can be added in the future (cf. section 4.5).

As also part of this *Infrastructure* layer and complementary to these model querying facilities, MoDisco integrates the use of dynamic metamodel extension capabilities. This is particularly relevant during the Model Understanding phase when aiming to retrieve and represent useful extra-information on discovered models. These extensions, called "facets", are computed at runtime and allow adding useful information to already existing models without actually modifying them. Initially developed within MoDisco, the Facet mechanism and tooling have been externalized and are now provided by the dedicated EMF Facet project [58].

Finally, MoDisco offers some generic support for facilitating the representation of metrics computation results (e.g., from legacy system's models) and for automatically generating some base visualizations out of them. As an entry point to this feature, MoDisco proposes a very simple metrics metamodel to express such results as *metrics* models (basically sets of Metric elements, each one of them having several Value elements that can be of different types). Then a predefined chain of model transformations allows, from such *metrics* models, to target different visualization formats. Currently, the available component covers the generation of simple HTML pages (displaying metrics results as tables), Excel sheets or SVG graphics (bar charts and pie charts).

4.3. MoDisco Technologies Layer

As part of its legacy technology dedicated support, MoDisco already provides a set of useful deployable components. These components are directly composable (by exchanging models via the Discovery Manager and Workflow for instance, cf. subsection 4.2.2)

with the ones from the infrastructure in order to address concrete reverse engineering scenarios. However, they are themselves independent from any specific use case. MoDisco currently offers ready-to-use support for three different legacy technologies, namely Java, JEE (including JSP) and XML. This list is going to be extended in the future by new research or industrial contributions covering other languages (e.g., CSharp) or technologies (e.g., databases).

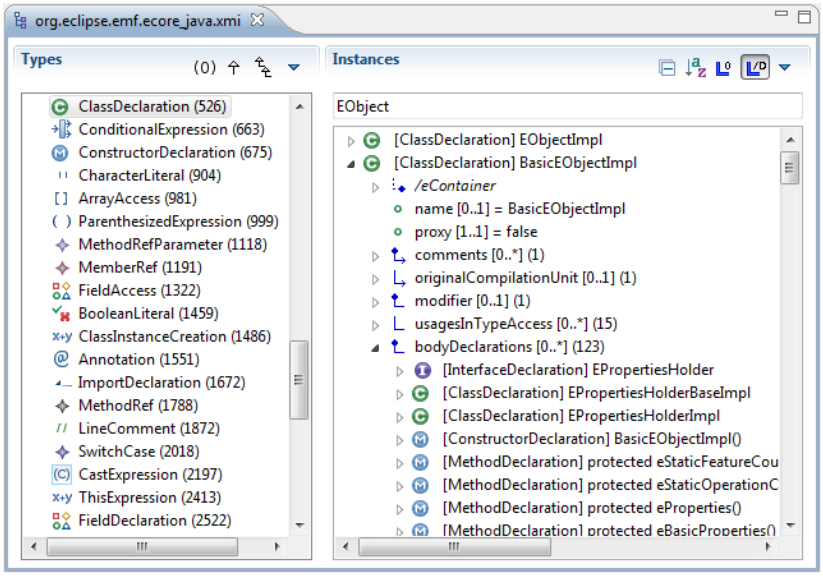

Figure 8: Example of a discovered Java model viewed in the MoDisco Model Browser

The Java dedicated features show the capabilities of the generic MoDisco framework by applying them in the context of a widely used technology such as Java. A complete Java metamodel based on the JDT, i.e., covering the full abstract syntax tree (AST) of Java programs from the package and class declarations to the method bodies' expressions and statements (that are also modeled in detail), is offered by the framework. Relying on this metamodel (cf. some examples of metamodel classes in the left panel of the model browser on Figure 8), a corresponding discoverer is available allowing to automatically obtain complete Java models out of any Java projects (cf. also Figure 8). As previously introduced in Section 3.2 when describing the *injection* step, the Java model discoverer uses a dedicated technology, JDT and more particularly its in-memory representation of the Java sources, as the technical solution to navigate the program AST (an associated visitor implementing the actual building of the Java model). The automated regeneration of the Java source code from these handled (and in the meantime possibly modified) Java models is also ensured thanks to a specific code generator implemented with a model-to-text dedicated technology (Acceleo [14]). A complementary model transformation permits the KDM discoverer directly producing generic KDM "Code" models (cf. Subsection 4.2) from such Java models. Still related to KDM, an additional discoverer is provided to automatically get composite (trace) models integrating both the Java and KDM "Source" elements (cf. Subsection 4.2) from any Java project.

Given the widespread use of XML documents in (legacy) software systems, a full and generic XML support is natively provided by MoDisco. A complete XML metamodel has been implemented conforming to the subset of the related W3C specification defining the XML core concepts: root, elements, attributes, etc. Thus, this metamodel is schema-independent and can be used to model any XML file, i.e., both XML documents and XML schema definitions (XSDs). To concretely allow this, the associated model discoverer is made available. This notably prevents from having to implement a particular discoverer for each XML-based file type, thus saving some useful effort. If really needed, an additional model transformation can still be built quite systematically from this generic XML metamodel to a given specific metamodel (cf. the Model Discovery two-step approach introduced in Section 3.2).

A specific support for JEE technologies has also been developed. This includes notably a metamodel for the JSP language (extending some core concepts from the XML one) as well as the corresponding discoverer allowing to get complete JSP models out of JSP source code. In addition to this JSP support, the JEE dedicated components cover the automated modeling of the most common JEE Web application configuration files (namely *web.xml* and *ejb-jar.xml*) via specific model discoverers. Finally, there is a predefined set of ready-to-use queries and facets for either highlighting existing JEE-specific information or extracting new JEE-related data from previously discovered Java models.

4.4. MoDisco Use Cases Layer

Complementary to all these components and to illustrate their actual use in MDRE processes, the MoDisco *Use Cases* layer offers some more features for specific reverse engineering scenarios.

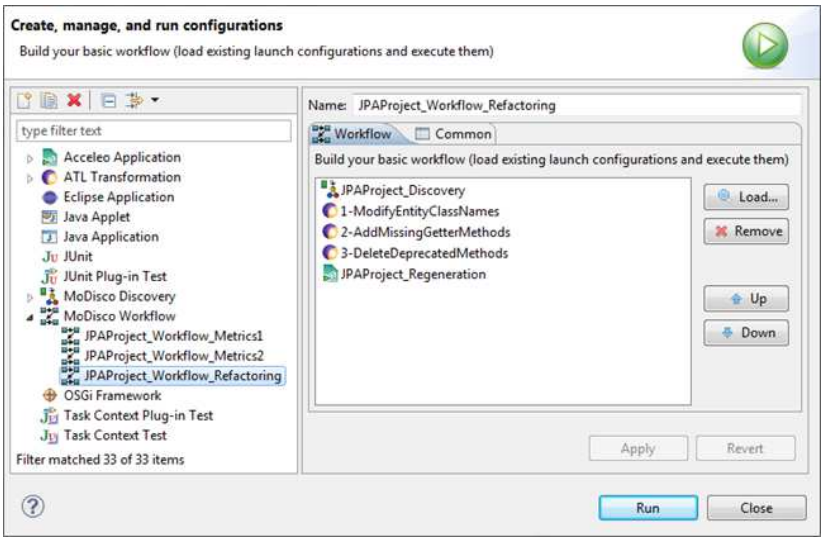

Figure 9: Generic MoDisco Worflow, with a sample Java refactoring process

MDRE processes have the natural capability to be largely automated. Thus, the different MoDisco components can be combined together to address particular reverse

engineering use cases. As mentioned before in subsection 4.2.2, MoDisco offers a dedicated workflow support specifically intended to the chaining of model discoveries with consecutive model transformations, related scripts, final code generations, other programs, etc. This comes with a dedicated window to select and order the operations (calling to MoDisco components or others) to be part of the given MDRE workflow.

As concrete examples, complete automated MDRE workflows are provided for instance to recover the design of Java applications or to perform some refactoring on Java code. The example in Figure 9 shows such a workflow composed of an initial Java model discover, three consecutive refining (model) transformations and a final Java code (re-)generation.

4.5. Extending MoDisco

MoDisco has been designed as a generic and extensible framework. There are different ways of extending it in order to allow a more in-depth or specific support to some MDRE scenarios or legacy technologies. Notably, as an Eclipse-based solution, the generic components come with extension points [59] and related interfaces that can be used to implement these more particular features.

The most straight-forward manner is to simply complement the *Use Cases* layer (cf. section 4.4) with other examples of MoDisco components' combinations (e.g., defining different workflows chaining these components) to address a given MDRE process.

Another direct way is to complement the *Technologies* layer (cf. section 4.3) by adding the support for another (legacy) language, framework, file format, etc. This implies first developing the corresponding metamodels (in EMF Ecore [20]), model discoverers, browser customizations, associated queries (e.g., in Java or OCL [52]) or transformations (e.g., in ATL [54]), generators (e.g., in Acceleo [14]), etc. For instance, for new model discoverers, a generic interface (*IDiscoverer*) has to be implemented and a specific extension point (*org.eclipse.modisco.infra.discovery.core.discoverer*) used so that the Infrastructure (the Discovery Manager in the present case) can then automatically identify and declare the component as part of the MoDisco environment. After that, such a newly added discoverer can be considered identically to other similar components provided with the base version of the framework.

The other way is to directly work on the *Infrastructure* level (cf. section 4.2). Apart from the addition of a new implementation of a standard metamodel (OMG ADM [23]), this is generally more complex and requires a deeper knowledge of the MoDisco internals and generic components APIs. For instance, advanced browser customizations (e.g., add new options, contextual actions or a different viewer) have to be implemented using the Model Browser specific API. Moreover, in addition to Java or OCL as already offered, new languages support can also be developed and plugged into the Query Manager. Another example of infrastructure extension is the possibility of supporting different workflow engines (than the provided base one) via the use of a dedicated extension point.

For getting more insights on all these technical aspects, please refer to the MoDisco Developer Documentation as available with any Eclipse MoDisco release.

4.6. Benefits of MoDisco

A clear separation of concerns in our MoDisco MDRE approach, combined with the generalized use of MDE during the full reverse engineering process, allows MoDisco to answer satisfactorily to the MDRE challenges identified in Section 2.

Firstly, the explicit distinction in the MoDisco architecture between technology and scenario-independent (*Infrastructure* layer), technology-specific (*Technologies* layer) and scenario-specific (*Use Cases* layer) components provides a high adaptability at two different levels: 1) the nature of legacy system technology and 2) the kind of reverse engineering scenario.

Secondly, the metamodel-driven approach followed in MoDisco enables covering different levels of abstraction and satisfying several degrees of detail depending on the needs of the reverse engineering scenario. All the required information can be actually represented as models so that there is no information loss during the MDRE processes (except for those details that the user explicitly wants to left out as part of a transformation process of the initial discovered models).

Thirdly, the use of MDE techniques allows the decomposition and automation of the reverse processes. They can be divided in smaller steps focusing on specific tasks, and be largely automated thanks to the appropriate chaining of corresponding MDE operations (notably model transformations). They can also directly benefit from the model exploration and extraction capabilities provided by these MDE techniques in order to improve legacy systems overall comprehension. All the involved modeling artifacts (models, metamodels, transformations, etc.) can be homogeneously re-used, modified for maintenance and evolution reasons, or extended for other purposes. Moreover, new transformations can be developed and plugged adding more capabilities without altering the already implemented features.

Finally, the treatment of the potentially huge amount of concerned data can be simplified because the models of the systems are the elements actually processed (thanks to the available modeling techniques) rather than directly the systems themselves (which are not modified during the process). The performance observed on the key MoDisco components are already acceptable for an industrial use in several concrete scenarios (cf. Section 5).

5. MoDisco MDRE Concrete Use Cases

Industrial reverse engineering scenarios typically involve different types of input artifacts like source code, documentation or raw data, as well as different kinds of resulting outputs such as new/modified source code or documentation but also models (i.e., views), metrics, etc. The MoDisco framework has already been applied and deployed in several of such real use cases with the underlying objectives of 1) ensuring its actual usability, 2) collect valuable feedback from users and 3) validating its capacity to manage the complexity of some industrial scenarios.

The various MoDisco components can be used by combining, chaining or integrating one or more of them into the MDRE solutions to be built. Thanks to the characteristics of the used Eclipse Public License (EPL) [60], in all cases the elaborated solutions can be fully open source, fully proprietary or follow an hybrid approach.

To illustrate the suitability of MoDisco in industrial scenarios, this section presents a couple of projects where the Mia-Software company built a MoDisco-based solution to solve real customer problems.

5.1. MoDisco Use Case 1: Java Application Refactoring

MoDisco has been used to handle a critical project concerning a legacy system of a big car rental company. The idea was to automate massive refactoring tasks on a large Java application (approximately 1000K source lines of code in total/SLOC) in order to improve both performances (notably in terms of memory usage) and code readability.

5.1.1. Process Overview

To be able to refactor the Java legacy system, specific patterns first needed to be identified in its source code. As a consequence, an initial reverse engineering phase was required in order to obtain an exploitable view (i.e., a model) of this system. Based on the discovered representation, the code upgrades have then been automatically performed at the model level (cf. Figure 10) by means of in-place model transformations. As a last step, the (upgraded) source code of the refactored Java application has been automatically generated from the modified models.

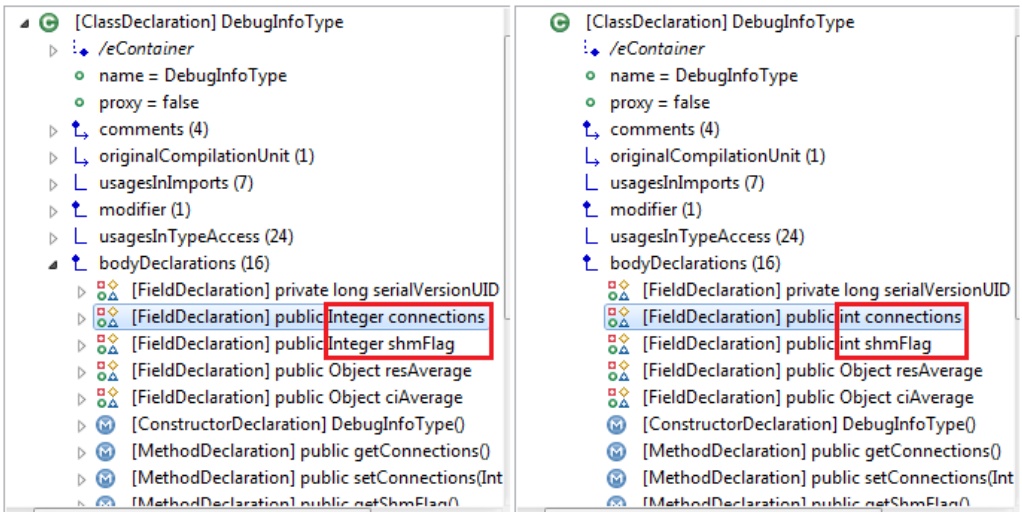

Figure 10: Java model before (left) and after (right) refactoring, using the MoDisco Model Browser to show the effects of the first refactoring

To perform the refactoring, three model transformations were used with the aim to:

- Replace the use of Java wrapping types (Integer, Double, Float, Long) by Java primitive types (int, double, float, long) in some identified parts of the code, as visible on Figure 10 (cf. also Figure 11 for an excerpt of the corresponding transformation rule);
- Clean the useless code calling to emphlog(based on the *Apache Commons Logging* framework);
- Delete some abusive uses of a specific client class (named *ValueHolder*, that was simulating some kind of C++ like pointers).

```
private void replaceBoxedFieldByPrimitiveType() {
    for (FieldDeclaration fieldDeclaration : allFieldDeclarations()) {
        if (isFormCode(fieldDeclaration) || isExcludedFromPrimitiveRules(fieldDeclaration)) {
            continue;
        }

        TypeAccess replacementForBoxedPrimitiveType = replacementForBoxedPrimitiveType(
                                                        fieldDeclaration.getType());
        if (replacementForBoxedPrimitiveType != null) {
            System.out.println("field declaration : replacing "
                    + fieldDeclaration.getType().getType().getName()
                    + " by "
                    + replacementForBoxedPrimitiveType.getType().getName()
                    + " in " + locate(fieldDeclaration));
            fieldDeclaration.setType(replacementForBoxedPrimitiveType);
            this.count2b++;
        }
    }
}
```

Figure 11: An example of Java application refactoring rule for type replacement

Note that these *refactoring* rules had to be applied on each of the different application releases.

This **model driven refactoring process** is an adaptation of the well-known Horseshoe model.

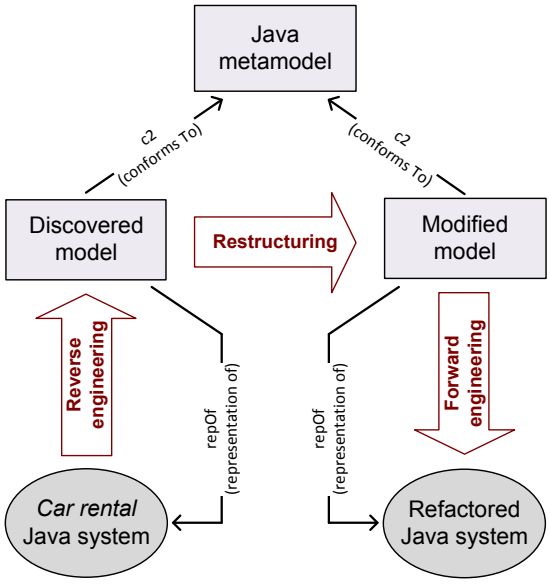

Figure 12: Model driven Java application refactoring: overall process

Figure 12 depicts its three main steps:

1. **Reverse engineering** from the input Java application by using the MoDisco Java metamodel and corresponding model discoverer (MoDisco *Technology* layer));
2. **Restructuring** of the obtained application model thanks to model transformations

in Java implementing the previously described rules (that could be positioned as part of the MoDisco *Use Cases* layer);

3. **Forward engineering** of the output (refactored) Java application from this modified model by running the MoDisco Java generator (MoDisco *Technology* layer)).

The choice of Java for implementing the model transformations (rather than using a dedicated model transformation language like ATL) has been made by Mia-Software because of their engineers' Java expertise. The MoDisco Model Browser (and Java metamodel specific customization) has been used for manual verification and testing at each step of the process and on of the several application model versions. The overall integration of the three different steps has been realized thanks to the provided MoDisco workflow facilities.

5.1.2. Observations

Globally, the use of MoDisco in the context of this refactoring use case has been a success. As a result of the process, approximately 60K SLOC from the whole application were concerned and so automatically refactored (Figure 13). An effective performance gain and overall readability improvement of the modified source code parts have then been confirmed on the regenerated application.

```
private static final long serialVersionUID = 1L;
public Integer connections;
public Integer shmFlag;
public Object resAverage;
public Object ciAverage;

//MIA tag : Start : user attributes of "DebugInfoType"
//MIA tag : End : user attributes of "DebugInfoType"
//Begin : Constructors of Class "DebugInfoType"
/**
* Builder DebugInfoType
*/
public DebugInfoType() {
    //MIA tag : Start : user modified constructor "DebugInfoType"
    //MIA tag : End : user modified constructor "DebugInfoType"
    connections = 0;
    shmFlag = 0;
    resAverage = 0;
    ciAverage = 0;
}
```

```
private static final long serialVersionUID = 1L;
public int connections;
public int shmFlag;
public Object resAverage;
public Object ciAverage;

//MIA tag : Start : user attributes of "DebugInfoType"
//MIA tag : End : user attributes of "DebugInfoType"
//Begin : Constructors of Class "DebugInfoType"
/**
 * Builder DebugInfoType
 */
public DebugInfoType() {
    //MIA tag : Start : user modified constructor "DebugInfoType"
    //MIA tag : End : user modified constructor "DebugInfoType"
    resAverage = 0;
    ciAverage = 0;
}
```

Figure 13: Sample Java code before (top) and after (bottom) refactoring

More specifically, and acccording to the Mia-Software engineers working on this

project, discovering intermediate model-based representations of the source code really improved their understanding of the application and also largely facilitated the elaboration of the transformations implementing the different modifications. Indeed, only 1 person/month (PM) has been required for realizing the full project, i.e., internally developing, then deploying and finally applying the MoDisco-based solution. In contrast, Mia-Software evaluated that following a semi-manual approach based on textual regular expressions (which is for them less reliable than a model-based approach) would have required at least 1 PM simply for realizing a study phase to identify the concerned parts of the application as well as needed expressions, and then again more to set up and perform the actual refactorings themselves. According to their effort calculation scheme (study phase + 500 SLOC/day/person for a manual processing), following a fully manual approach could have cost up to 7 PM for the same project.

Due to the relatively important size of the targeted legacy system, the main problem encountered during the process was scalability-related. This was linked to the use of the single EMF framework without other complementary (scalability) solutions such as CDO [61] at the time. A specific parameterization of the model discovery to filter out some Java packages that were not involved in the refactoring (via the corresponding Java model discoverer parameters), as well as the split of the loaded model into several derived models, helped solving this issue.

To summarize, using available MoDisco to implement the MDRE solution to this use case has allowed the company saving time and resources (and therefore reducing the project costs) for the following main reasons:

- The Java metamodel, corresponding injector and extractor were already provided for free and directly reusable. This permitted really focusing on the transformation part which is the core part in a refactoring process;
- The understanding of the handled application and writing of the model transformation rules (in Java in that case) have been practically facilitated and accelerated by both the well-structured Java metamodel and the model navigation capabilities the MoDisco Model Browser is offering;
- The general automation of the solution (including the developed transformations) has been made easier thanks to the MoDisco integrated framework and corresponding MDRE workflow support.

5.2. MoDisco Use Case 2: Code Quality Evaluation

MoDisco has been integrated in the Mia-Quality solution which is dedicated to the *quality* monitoring of existing applications. This software system has already been deployed several times, e.g., to verify the main product of an insurance management software provider.

5.2.1. Process Overview

To efficiently evaluate the quality of a given legacy application, an automated quality analysis process has to be put in place. However, all the (often heterogeneous) technologies combined in the input system have to be covered to obtain relevant results. This means that the quality solution to be designed and implemented has to be completely technology-agnostic, and to potentially deal with any kind of quality measures/metrics.

Thus, the solution needs to be parameterizable by both the measurements to be actually performed and the software to be monitored. Mia-Quality is a commercial product offering such a generic solution benefiting from the reuse and integration of several MoDisco components.

This **model driven quality evaluation process** is composed of three main steps (Figure 14):

1. **Measurement** by analyzing models of reverse engineered applications. These models can be potentially obtained thanks to MoDisco model discoverers (for Java, JEE, XML) and related transformations for instance, or via provided import capabilities compatible with other existing tools such as Checkstyle [62];
2. **Consolidation** by connecting external quality analysis tools and aggregating the data coming from these different tools in a unique quality measurement model. This integration is realized by using a copy of the MoDisco SMM metamodel, as the pivot metamodel in the quality solution, with related model transformations;
3. **Presentation** by displaying the quality analysis results in the specified formats according to user requirements. For instance, the external tool Sonar [63] has been plugged to the solution and customized for showing these results (i.e., the measurements model) in a graphical way.

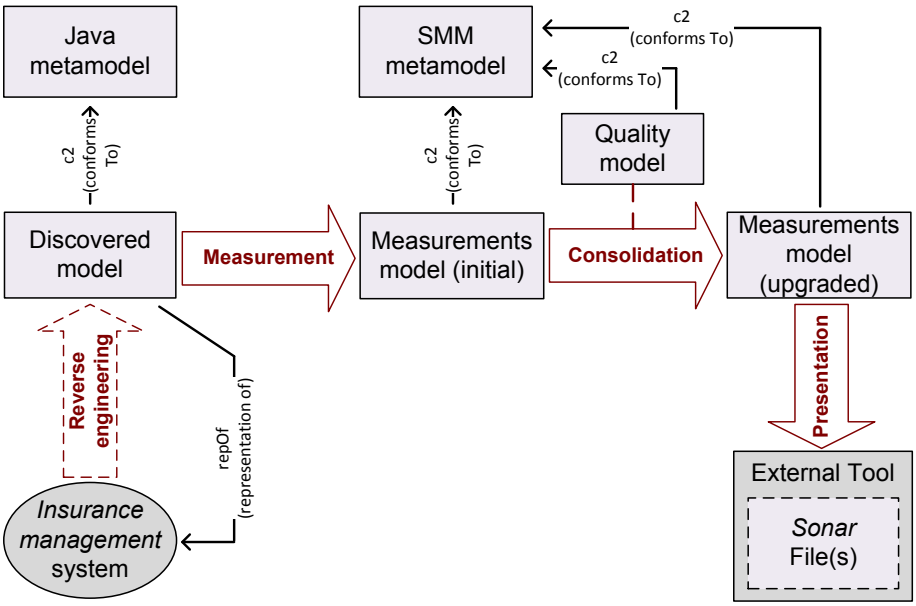

Figure 14: Model driven code quality evaluation: overall process

The MoDisco SMM metamodel is largely used in the solution for both specifying homogeneously the measures and representing the results of their computation on the application models. The data exchanged between the three steps of the process are almost exclusively SMM models. They are treated either automatically via Java code or manually thanks to a dedicated editor (Figure 15). This editor is based on parts of the

MoDisco Model Browser completed with additional specific customizations for the SMM metamodel.

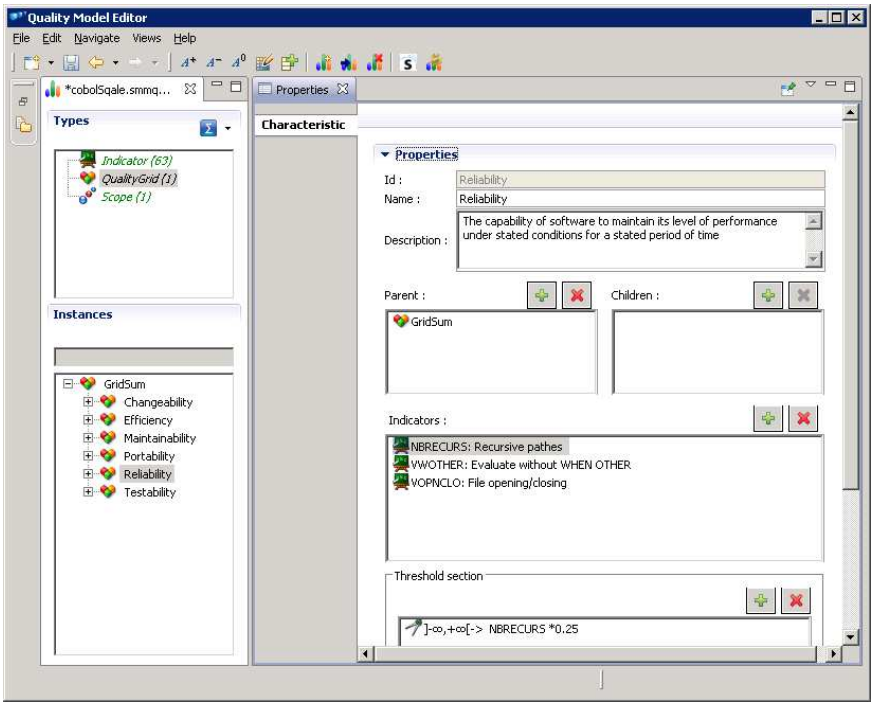

Figure 15: The MoDisco-based Mia-Quality Model Editor

5.2.2. Observations

In the context of this use case, several MoDisco components have been successfully combined as part of an automated quality evaluation solution. The built solution fits the expressed needs in terms of genericity and automation, and has also been concretely applied on the initially targeted insurance management software (as well as in other projects). Figure 16 shows samples of concrete final results as graphically displayed in Sonar.

Mia-Software clearly benefitted from the use of Modisco (and some of its preexisting components) in order to build faster the Mia-Quality solution. Moreover, although it is quite difficult to precisely evaluate actual benefits for end-users, Mia-Quality users also reported a positive feedback regarding the productivity and flexibility of the tool (thanks to the use of the MoDisco SMM Metamodel and the integration of several MoDisco Model Browser graphical features).

This use case has demonstrated the ability of MoDisco to be considered not only as an integrated framework (cf. Section 5.1), but also as an interesting provider of automatically reusable MDRE components. Thus, their genericity and customizability largely facilitate their integration both within and with other existing solutions.

To summarize, using (parts of) MoDisco to implement the MDRE solution to this use case was valuable for the following main reasons:

- The SMM metamodel, used as both the core metamodel inside the solution and the pivot one for integration with external tools, was already provided for free and directly usable. SMM is also a recognized interoperability standard promoted by the OMG;
- The various MoDisco model discoverers can be reused to provide different kinds of input to the code quality evaluation solution (model discovery phase) (but note that only some importers from external tools are distributed in the commercial version so far);
- Several graphical components from the Model Browser (e.g. tree viewer, customization support) have been used as the basis for building the solution's Quality model editor (model understanding phase) as shown on Figure 15, allowing considerably reducing the required development effort.

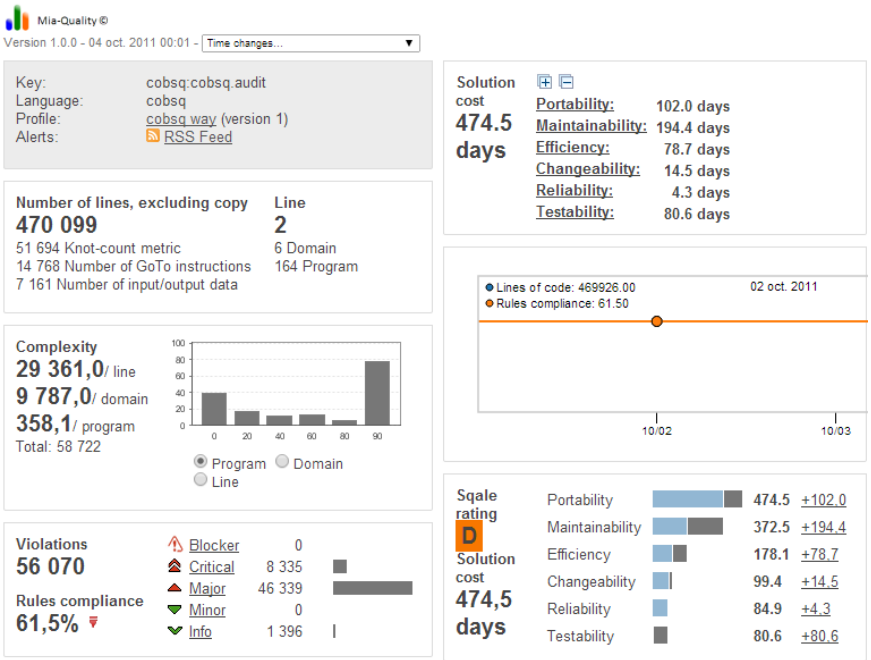

Figure 16: Some quality measurements obtained as output of Mia-Quality

6. Performance Evaluation

Scalability is one of the MoDisco (and MDRE) main challenges as stated in Section 2. Thus, this section presents performance evaluations highlighting the ability of MoDisco to

be useful in industrial scenarios where models go far beyond the toy examples frequently used in research papers. The purpose is to evaluate MoDisco's performance according to several fundamental criteria: execution time, memory footprint, model size, etc. The results show that MoDisco is a suitable candidate for companies looking for reusable components to build their MDRE solutions.

The evaluations focus on one of the key components in any MDRE solution when facing scalability issues, the model discoverer. In particular, we analyze the results obtained from the automated tests realized on the MoDisco Java model discoverer [5]. The experiments have been realized on a standard development machine with the following configuration: Quad Core processor at 2.40GHz (Intel), 4GB of RAM, x86 architecture (Windows 7 OS). The measurement components have been implemented by using the Java standard API. The used memory is computed as the difference at a given time between the total memory (*Runtime.getRuntime().totalMemory()*) and the still free one (*Runtime.getRuntime().freeMemory()*). The required time is obtained via calls to *System.getTime()* at both the beginning and end of (the part of) the process to be measured. Due to some system or Java internals, a measurement overhead can be sometimes observed.

Globally, the following four performance indicators have been considered by the executed evaluations:

- Size of the discovered Java models;
- Memory footprint during the discovery process;
- CPU time needed for performing this same process;
- Internal repartition of effort inside this process.

6.1. Experiment 1: discovered models size

As input for this experiment, we have used several Java legacy systems (plug-ins from the Eclipse Platform in this case) of increasing sizes measured in number of lines of code (LOC). Note that the first (small) systems have been considered only for evaluation purposes, MDRE and MoDisco becoming actually valuable when applied on systems of medium size or more. Also, the computation of a comment rate in the code could have allowed to evaluate a bit more precisely their actual size. However, the raw number of LOC has been retained as a sufficient indicator for the evaluation.

Figure 17 shows the results of evaluating the size of the discovered Java models for each plug-in. The number of model elements (in this context *objects* in the model) and the memory used on hard drive disk after XMI serialization [64] have been computed for each of them.

As expected, both the number of model elements and the size of the serialized models grow quite fast when the input application gets larger (in LOC). Nevertheless, the discovered models size stays relatively proportional to the input legacy systems size (with

[5] Different implementations have already been provided in MoDisco for the Java discoverer, all based on EMF. One implementation just uses the standard EMF API while another relies in addition on the CDO framework [61] dedicated to the handling of very big models. However, the evaluations presented here only concern the standard EMF-based implementation of the Java discoverer.

Eclipse Plug-ins	Lines of code	Number of model elements	XMI model size in Megabytes
org.eclipse.jdt.apt.pluggable.core	781	3,449	0.807
org.eclipse.jdt.apt.ui	2,113	10,217	2.541
org.eclipse.jdt.compiler.tool	2,195	10,187	2.476
org.eclipse.jdt.compiler.apt	6,885	29,444	7.639
org.eclipse.jdt.launching	12,172	52,205	12.801
org.eclipse.jdt.apt.core	13,854	59,270	14.723
org.eclipse.jdt.junit	14,744	66,411	16.206
org.eclipse.jdt.debug	39,411	156,374	38.432
org.eclipse.jdt.debug.ui	39,526	159,028	42.326
org.eclipse.jdt.core	278,045	1,430,345	367.828
org.eclipse.jdt.ui	325,657	1,444,753	393.959

Figure 17: Benchmark on the size of discovered Java models

a multiplier between 4 and 5 for the number of model elements, and between 0.0010 and 0.0013 for the XMI size). With a small input project of less than 800 LOC, the generated model contains only 3,500 model elements for a size inferior to 1MB when serialized. A more consistent application of around 40,000 LOC is already represented by more than 155,000 model elements and 42MB of XMI data. Obviously, when considering a larger project (325,000 LOC), the number of model elements becomes huge (almost 1,450,000) as well as the size of the corresponding XMI file (almost 400MB).

This behavior highlights the predictable scalability issues when dealing with even larger legacy systems and their models. The number of model elements is very important because it implies that the available memory could not be sufficient in some case to actually load the full models. Thus, we are currently exploring the possible use of optimization techniques such as the lazy loading of elements at the model manipulation time (e.g., during a transformation). The serialization (or more generally storing) size is also fundamental since it may limit the loading and saving of big models. Several alternatives (e.g., in Eclipse) have been proposed relying on different storage environments, and notably databases. As a consequence, we have already started looking to the integration of solutions like CDO [61].

6.2. Experiment 2: time vs. memory footprint of a discovery process

This evaluation and the next one focus on the two largest Java projects from Figure 17. They are actually the most relevant inputs according to the size of the applications to be normally tackled with MoDisco in realistic MDRE scenarios. Thus, the required time and memory footprint (i.e., used RAM) have been evaluated while executing full Java discovery processes on these two examples. Note that the evaluation itself is also using some memory during the discovery process. Nevertheless, this amount of memory is not significant compared to the total memory used, and so can be voluntarily ignored when analyzing the obtained results as graphically displayed on Figure 18.

The full discovery from the *org.eclipse.jdt.core* Java project takes more than 250,000ms (i.e., nearly 4 minutes) and needs at most 350MB in memory. The discovery from the larger *org.eclipse.jdt.ui* project has been performed in around 400,000ms (almost 7 minutes) and has used more than 500MB of memory. Considering the nature and frequency

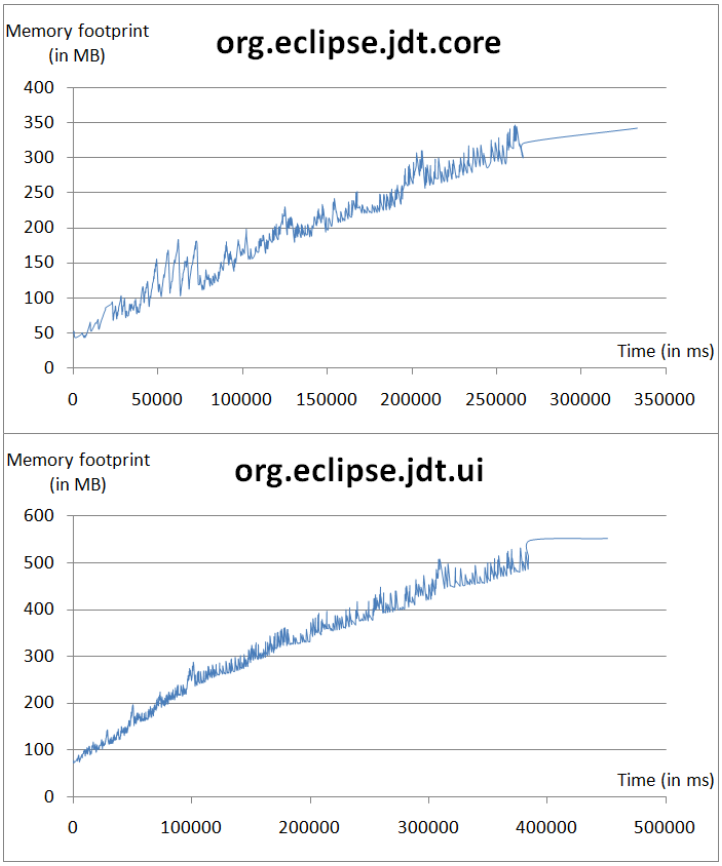

Figure 18: Benchmark on the time and memory footprint of a Java discovery process

of a classical discovery process, the total execution times are in both cases reasonable according to the inputs size.

The potentially important memory footprint (notably at the end of the process) could imply scalability issues when using standard computers. However, this in-memory footprint is interestingly proportional to the time. This could allow anticipating the amount of dynamically allocated memory for a given discovery process, depending on the input legacy application size and also current process running time. This knowledge could be particularly useful when working on optimizing long discovery processes from very large legacy systems.

6.3. Experiment 3: time repartition during a discovery process

Finally, a last experiment on the two same examples has focused on the internal effort repartition during a full Java discovery process.

The objective was to collect interesting information on the required time per different subtask inside such a discovery process. In that specific case, these subtasks are: 1) creation of the abstract syntax tree from the Java program, 2) visit of this tree to

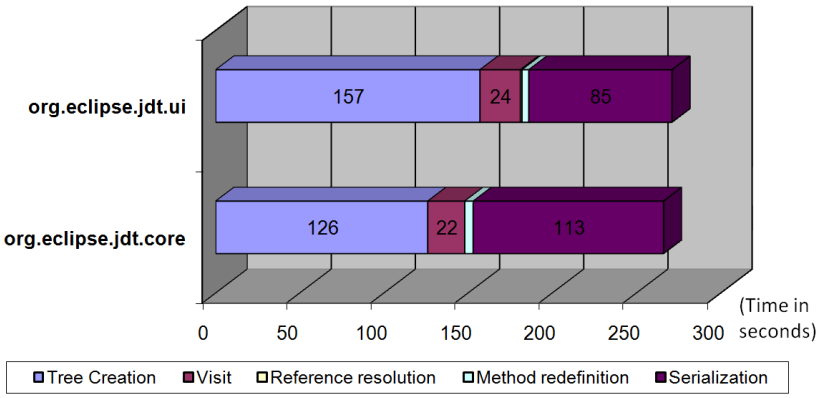

Figure 19: Benchmark on the time repartition during a Java discovery process

generate the corresponding model elements, 3) resolution of the references between the different created model elements, 4) redefinition of some methods when required and 5) serialization in XMI of the built model. Note that the provided total discovery times are approximations, advanced reference resolutions having been voluntarily skipped. In any case, the time allocated to this reference resolution largely depends on the needed degree of details and could allow interesting scenario-specific performance optimizations. The obtained results are graphically presented in Figure 19.

This evaluation generally emphasizes that the Java program abstract syntax tree (AST) creation and discovered Java models serialization are very time consuming tasks. It also denotes that the actual visit of the AST (including both the navigation and production of model elements) is relatively fast compared to them. This highlights two main discovery process parts where considerable performance gains could be obtained. This last experiment provides precious indications on where to significantly improve the overall discovery process scalability (cf. Section 9).

7. Project Features and Statistics

This section lists the main items provided by the MoDisco project and gives some concrete numbers about its size and activity level.

MoDisco comes with its dedicated website under *Eclipse.org* [55]. Moreover, a MoDisco specific section in the Eclipse Wiki is regularly updated with the latest project news and important information [65]. This Wiki section is also the main source of documentation for the project in general, e.g., concerning the different MoDisco releases, in addition to their official documentation [66]. As an open source project, a direct access to the source code repository is provided to everybody (in a read-only mode for non-committers). Note that, following the current Eclipse policy, the migration from SVN to Git has been recently performed. A MoDisco forum is publicly available for any kind of project related discussions, and a dedicated mailing list is also usable for the project developers, i.e., for both official committers and external contributors (*modisco-dev@eclipse.org*). MoDisco is completely integrated, as an Eclipse project, in the open Eclipse Bugzilla: both developers and users can have access to the related issues, requests for features, etc. and of

course raise their own problems if necessary.

Thanks to this overall infrastructure, the activity of the project can be measured and evaluated using real figures. Thus, all the numbers provided hereafter are publicly available via the project's repository, documentation or forum, the Eclipse Bugzilla, etc.

The various MoDisco committers, contributors and users are constantly active as visible from the Eclipse Bugzilla (Figure 20). At the 18th of February 2014, 884 MoDisco *bugs* were entered in the system: they can be for raising issues, providing concrete patches, proposing new contributions to be integrated or simply presenting feature requests on the three different MoDisco project layers. More generally, they are used to follow and keep track of the different steps of the Eclipse project management and development processes. An important part of these *bugs* have been actually addressed (611 bugs marked as fixed in total, for a resolution rate of almost 70%), the others being progressively processed according to their assigned priority.

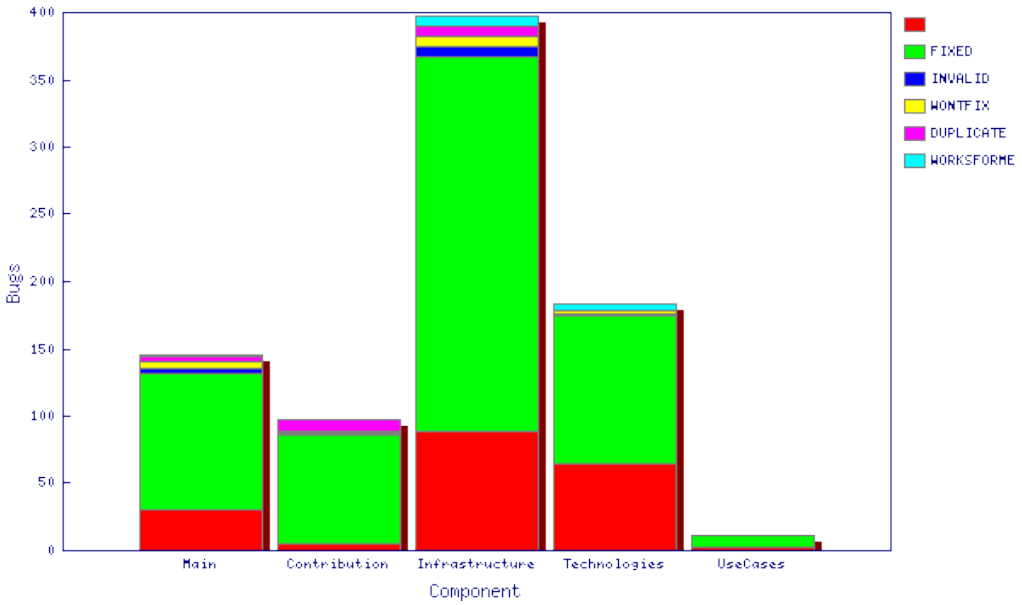

Figure 20: MoDisco *bugs* in Eclipse Bugzilla (green/red means already/to be addressed)

As a consequence, the MoDisco source code is also regularly upgraded and new components or capabilities added. Since the first MoDisco official release (within the Eclipse Helios Simultaneous Release in June 2010), more than 1610 commits have been done on the MoDisco repository containing the MoDisco plugins sources (and previously 3,263 commits just for this initial MoDisco Helios release). These latest commits concern a total of more than 719,000 lines of code over more than 2,047,000 available from this same repository. The last past year, the effort has been mainly put on stabilizing the existing MoDisco components (bug fixing, performance improvement, minor enhancement, etc.).

Externally, the MoDisco project is progressively getting more visible and its community growing accordingly. The forum is very regularly populated by questions and comments from MoDisco users as well as general news from committers. It is the place

for interactions between the different MoDisco community members (more than 245 different discussions for more than 955 posts in total on the project forum). Another remarkable sign of the community evolution is the number of downloads of the various MoDisco builds. Thus the different MoDisco released versions (i.e., Helios in June 2010, Indigo in June 2011, Juno in June 2012, Kepler in June 2013 and coming Luna in June 2014) have been downloaded more than 78,000 times in total (including the SR1 and SR2 releases for each of them).

8. Lessons Learned

Creating, developing and disseminating the MoDisco framework (i.e., both the approach and related tooling) during the last past years has allowed us to identify several key factors that could also be relevant in the context of many other software development projects:

- **Collaborating with an industrial partner.** Companies often consider too risky to use research prototypes due to their lack of proper user interface, documentation, completeness, support, etc. Research groups have limited resources, therefore these resources cannot be usually devoted to work on such non-core research activities and these prototypes remain simple proofs-of-concept. Thus research groups can miss the opportunity of having a large tool user base, along with its associated benefits (e.g., empirical validation, feedback, visibility, collaboration opportunities).

  To avoid this "fate" for MoDisco (that started as a research project), we decided to partner with a technology provider (Mia-Software, part of the SODIFRANCE Group) to industrialize MoDisco. This allows the research group to focus on doing research while staying sure to work on actually relevant industrial challenges. Instead, the industrial partner takes over traditional software development and maintenance tasks. This strengthens its presence and visibility within the Eclipse community and surrounding open source market, and is also used to create a network to expand its business (e.g., by providing training or special consulting services). However, even if such a partnership has worked well in the case of MoDisco, we have sometimes failed reproducing the approach with other initiatives. This has highlighted the difficulty of finding the appropriate company, setting up the proper process and more than all getting the required initial funding to support such a collaboration.

- **Using an open source license**. The global choice of adopting an open source approach for publishing the tool, managing it and communicating around the different project results has been fundamental. It simplifies a lot all common actions among the partners, notably concerning legal aspects (such as the intellectual property) or the exploitation and dissemination of the results (specially for the research team). The fact that the selected open source license, the Eclipse Public License (EPL) [60], allows commercial adaptations and redistributions of the software is also a win-win situation since it can help to convince companies to participate and contribute to the MoDisco open source code base while allowing them to later on adapt the components for commercial use.

Open source is gaining more and more credibility, particularly in the industrial world. But there is still some reluctance, especially in big companies, to really integrate this new business model. In this context, innovative SMEs can be easier to convince as open source provides some kind of flexibility quite interesting for them.

- **Integrating a widely recognized community**. Open source is not enough per se to attract new users as many open source projects are half-dead, of relatively poor quality or adopted by a very limited set of users. Instead, being an official Eclipse project has given a lot of visibility to MoDisco. The Eclipse brand is very well-known and considered as a guarantee of high-quality by many practitioners.

  Thus, the effective integration of MoDisco into such an industrially recognized project/community has a very beneficial impact over the general perception of the tool by IT professionals. We recommend to try identifying such a well-known community in its domain to benefit from the visibility and interactions with its members. However, this is a bidirectional effort: the research team also needs to invest on the community. In our case, for instance, we attend the Eclipse conferences and try to participate to and use other Eclipse projects beyond MoDisco.

- **Following structured development processes**. When building a tool as complex as MoDisco, you need to have a well-defined development process (milestones, bug tracking, version control, tests, coding style guidelines, etc.). We were not following all these best practices at the beginning, but the growing complexity of the project and increasing number of users forced us to adopt them. Also, the released code being open source, the work of the development team had to be better and the reliability of the built solution improved.

  Being part of Eclipse, with its strict procedures [67], really helped in this matter. In addition, the Eclipse Simultaneous Release yearly cycle forces the project team to release and update tool versions (and related documentation) on a regular basis. However deploying such a process can be quite heavy and so generally cannot be supported by a research group alone, making the partnership with a company a requirement.

- **Relying on a reference framework (i.e., EMF)**. For a tool to be stable and reliable, it must be built on solid ground. Thus, being based on an already well-established and recognized framework is a guarantee of a certain quality level. Moreover, this also helps targeting a more widespread audience (due to the shared use of a common framework). In the context of MoDisco, the choice of using EMF as the reference modeling framework has been quite obvious since EMF was and is still today the most widely use (open source) framework of this type.

  However, while capitalizing on the numerous benefits brought by EMF, we also inherit from his drawbacks. Currently, EMF still has some scalability issues when dealing with very big models and these are open spaces for research and experiments (cf. Section 9). But globally, such drawbacks are compensated by the advantages of relying on a mature framework like EMF.

9. Conclusion and Future Work

The number of software systems to be maintained, extended or generally evolved has grown considerably during the last decades and will continue to do so. In order to deal with all this legacy software, both economically and technologically speaking, reliable (semi-)automated reverse engineering solutions must be provided. To reach this objective, integrating MDE techniques in reverse engineering solutions shows promising and innovative results.

In this paper, we have presented MoDisco that offers a generic and extensible model-driven reverse engineering (MDRE) framework intended to facilitate the elaboration of MDRE solutions actually deployable within industrial scenarios. The provided description includes its overall underlying approach, architecture, available components and the detail of its two main reverse engineering phases, namely *Model Discovery* and *Model Understanding*. Real applications of the MoDisco framework on concrete industrial use cases have also been presented as well as performance benchmark results, general project statistics and the main lessons learned from our overall experience.

On one hand, MoDisco as a project has developed significantly since its creation. Thanks notably to the active support of Mia-Software (Sodifrance), it has grown from a research initiative to an industrialized project having a quite solid user base and regular stable releases. The project is continuously open to requests, enhancements or new contributions either by using the Eclipse infrastructure tools such as the forum, Bugzilla, etc. or by directly contacting the project team. But getting external people or new partners to actually join in such an open source project, with the long-term involvement that this implies (e.g., maintenance or release engineering tasks during several years), is not something easy. In fact we have received concrete requests for contributions several times in the past years (e.g., new model discoverers from different legacy technologies), but only few of them actually pass all the required steps to be finally integrated in the various MoDisco official builds (such as done for the GASTM and SMM standard metamodel implementations, respectively contributed initially by Open Canarias and Castor Technologies).

On the other hand, the MoDisco framework as such has been successfully deployed or reused in different industrial MDRE scenarios (cf. Section 5 for some concrete examples), thus showing its capabilities in terms of **adaptability/portability** and **no loss of required information**. However, we plan to continue testing the extensibility and improving the **coverage** of the framework by applying it on other technologies: e.g., some experiments around the migration of C#/.NET systems are being performed within the context of the ARTIST FP7-ICT European project [68]. Moreover, some MoDisco components are being progressively externalized to facilitate their reuse in other projects (which are not necessarily dealing with reverse engineering). The most remarkable example has been the creation of the Eclipse EMF Facet project [58] as a spin-off from MoDisco.

Other future research lines include the provision of more advanced model **understanding** components able to provide higher abstract views to improve **system comprehension** (notably by domain experts). In particular, we are currently exploring the (semi-)automated extraction of the business rules the legacy system is actually enforcing so that the organization can validate they are still aligned with the organization policies.

A considerable effort is also devoted to the general **scalability** challenge in order to

efficiently deal with the very large models typically obtained when reverse engineering large code bases. The current version of MoDisco has already proved to work on real projects of small and medium size (according to an industrial scale). However, more work has to be done to improve the performance when tackling very large systems. In this sense, we plan to experiment on various techniques such as notably lazy loading (to minimize the in-memory footprint, a model element is only loaded when requested by the user) and model transformation parallelization (to reduce the execution time, transformation rules are executed in parallel when possible).

10. Acknowledgements

The MoDisco project has been initially created within the context of the MODELPLEX FP6-IST European project, current research and developments are partially supported by the ARTIST FP7-ICT European project [68]. The information on the MoDisco use cases, benchmarks and project's general statistics has been collected with the precious help of Fabien Giquel and Nicolas Bros, two MoDisco committers from Mia-Software.

References

[1] M. Rekoff Jr., On reverse engineering, IEEE Transaction on Systems, Man and Cybernetics 15 (1985) 13–17.
[2] E. J. Chikofsky, J. H. Cross II, Reverse engineering and design recovery: A taxonomy, IEEE Software 7 (1990) 13–17.
[3] G. Canfora, M. Di Penta, L. Cerulo, Achievements and challenges in software reverse engineering, Communications of the ACM 54 (2011) 142–151.
[4] H. Chapman, P. A. Hall, Software Reuse and Reverse Engineering in Practice, Chapman and Hall, Ltd. London, UK, UK, 1992.
[5] M. L. Nelson, A survey of reverse engineering and program comprehension, in: In ODU CS 551 Software Engineering Survey, p. 2.
[6] G. Canfora, A. Cimitile, M. Munro, Re2: Reverse-engineering and reuse re-engineering, Journal of Software Maintenance: Research and Practice 6 (1994) 53–72.
[7] W. Premerlani, M. Blaha, An approach for reverse engineering of relational databases, in: Proceedings of the Working Conference on Reverse Engineering, Baltimore, Maryland, USA, May 21 - 23, 1993, pp. 151–160.
[8] E. Eilam, Reversing: Secrets of Reverse Engineering, Wiley Publishing, 2005.
[9] S. Kent, Model driven engineering, in: Integrated Formal Methods, volume 2335 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 286–298.
[10] H. Bruneliere, J. Cabot, F. Jouault, F. Madiot, Modisco: a generic and extensible framework for model driven reverse engineering, in: Proceedings of the ASE '10 IEEE/ACM International Conference on Automated Software Engineering. Antwerp, Belgium, September 20 - 24, 2010, ACM New York, NY, USA, 2010, pp. 173–174.
[11] G. Barbier, H. Bruneliere, F. Jouault, Y. Lennon, F. Madiot, Information Systems Transformation: Architecture-Driven Modernization Case Studies, The Morgan Kaufmann/OMG Press, pp. 365–400.
[12] J. Bézivin, On the unification power of models, Software and Systems Modeling (SoSyM) 4 (2005) 171–188.
[13] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, Science of Computer Programming: Special Issue on Second issue of experimental software and toolkits (EST) 72 (2008) 31–39.
[14] Eclipse acceleo project, `http://eclipse.org/acceleo`, Accessed February 17, 2014.
[15] Omg model driven architecture (mda), `http://www.omg.org/mda`, Accessed February 17, 2014.
[16] Omg meta object facility (mof), `http://www.omg.org/mof`, Accessed February 17, 2014.

[17] Omg unified modeling language (uml), www.uml.org, Accessed February 17, 2014.
[18] Omg query/view/transformation (qvt), http://www.omg.org/spec/QVT, Accessed February 17, 2014.
[19] Omg mof model to text transformation language (mofm2t), http://www.omg.org/spec/MOFM2T, Accessed February 17, 2014.
[20] Eclipse modeling framework (emf) project, http://www.eclipse.org/modeling/emf, Accessed February 17, 2014.
[21] S. Rugaber, K. Stirewalt, Model driven reverse engineering, IEEE Software 21 (2004) 45–53.
[22] J.-M. Favre, Foundations of model (driven) (reverse) engineering : Models - episode i: Stories of the fidus papyrus and of the solarus, in: J. Bezivin, R. Heckel (Eds.), Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings, IBFI, Dagstuhl, Germany, 2005.
[23] Omg architecture driven modernization (adm), http://adm.omg.org, Accessed February 17, 2014.
[24] L. Favre, Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution, IGI Global - Premier Reference Source, 2010.
[25] T. Girba, The Moose Book, Self Published, 2010.
[26] A. van Deursen, E. Visser, J. Warmer, Model-driven software evolution, in: 1st Workshop on Model-Driven Software Evolution (MoDSE 2007), Amsterdam, the Netherlands.
[27] K. Smolander, K. Lyytinen, V.-P. Tahvanainen, P. Marttiin, Metaedit a flexible graphical environment for methodology modelling, in: International Conference on Advanced Information Systems Engineering (CAiSE'91), Springer, 1991, pp. 168–193.
[28] J.-P. Tolvanen, M. Rossi, Metaedit+: Defining and using domain-specific modeling languages and code generators, in: International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03), ACM, 2003, pp. 92–93.
[29] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi, The generic modeling environment, in: International Workshop on Intelligent Signal Processing (WISP'2001), IEEE, 2001.
[30] L. Baresi, M. Pezze, A toolbox for automating visual software engineering, in: Fundamental Approaches to Software Engineering (FASE 2002), Springer, 2002, pp. 189–202.
[31] H. M. Sneed, Migration of procedurally oriented cobol programs in an object-oriented architecture, in: International Conference on Software Maintenance (ICSM 1992), Orlando, Florida, U.S.A.
[32] H. M. Sneed, Migrating from cobol to java, in: International Conference on Software Maintenance (ICSM 2010), Timisoara, Romania.
[33] F. Barbier, S. Eveillard, K. Youbi, E. Cariou, Model-driven reverse engineering of cobol-based applications, in: R. Vogel (Ed.), Proceedings of the Tools and Consultancy Track. 5th European Conference on MDA Foundations and Applications (ECMDA-FA 2009). Enschede, The Netherlands, June 23 - 26, 2009, pp. 36–51.
[34] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, J.-M. Jézéquel, Model-driven engineering for software migration in a large industrial context, in: Model Driven Engineering Languages and Systems, volume 4735 of *Lecture Notes in Computer Science*, 2007, pp. 482–497.
[35] M. Clavreul, O. Barais, J.-M. Jézéquel, Integrating legacy systems with mde, in: International Conference on Software Engineering (ICSE'10), IEEE/ACM, 2010, pp. 69–78.
[36] A. Alnusair, T. Zhao, Towards a model-driven approach for reverse engineering design patterns, in: U. A. Fernando Silva Parreiras, Jeff Z. Pan (Ed.), Proceedings of the 2nd International Workshop on Transforming and Weaving Ontologies in MDE (TWOMDE 2009). Denver, Colorado, USA, October 4, 2009, volume 531.
[37] O. Sánchez Ramon, J. Sánchez Cuadrado, J. García Molina, Model-driven reverse engineering of legacy graphical user interfaces, in: Proceedings of the ASE '10 IEEE/ACM International Conference on Automated Software Engineering. Antwerp, Belgium, September 20 - 24, 2010, ACM New York, NY, USA, 2010, pp. 147–150.
[38] R. Ferenc, A. Beszedes, M. Tarkiainen, T. Gyimothy, Columbus – reverse engineering tool and schema for c++, in: International Conference on Software Maintenance (ICSM 2002), IEEE Computer Society, 2002, pp. 172–181.
[39] F. Heidenreich, J. Johannes, M. Seifert, C. Wende, Closing the gap between modelling and java, in: International Conference on Software Language Engineering (SLE'09), Springer, 2009, pp. 374–383.
[40] Spoonemf2, http://soft.vub.ac.be/soft/research/mdd/spoonemf2, Accessed February 17, 2014.
[41] L. C. Briand, Y. Labiche, J. Leduc, Toward the reverse engineering of uml sequence diagrams for distributed java software, IEEE Transaction on Software Engineering 32 (2006) 642–663.
[42] W. Sun, S. Li, D. Zhang, Y. Yan, A model-driven reverse engineering approach for semantic web

services composition, in: Proceedings of the 2009 WRI World Congress on Software Engineering (WCSE 2009). Denver, Colorado, USA, October 4, 2009, volume 3, IEEE Computer Society Washington, DC, USA, 2009, pp. 101–105.
[43] F. Deissenboeck, L. Heinemann, B. Hummel, E. Juergens, Flexible architecture conformance assessment with conqat, in: International Conference on Software Engineering (ICSE'10), IEEE/ACM, 2010, pp. 247–250.
[44] J. Ebert, B. Kullbach, V. Riediger, A. Winter, Gupro - generic understanding of programs an overview, in: First International Conference on Graph Transformation (GraBaTs 2002), Elsevier, 2002, pp. 47–56.
[45] B. Roy, T. C. N. Graham, An iterative framework for software architecture recovery: An experience report, in: European Conference on Software Architecture (ECSA 2008), Springer, 2008, pp. 210–224.
[46] M. Lanza, Codecrawler - polymetric views in action, in: International Conference on Automated Software Engineering (ASE 2004), Linz, Austri.
[47] R. Wettel, M. Lanza, R. Romain, Software systems as cities: a controlled experiment, in: International Conference on Software Engineering (ICSE 2011), Honolulu, Hawaii, U.S.A.
[48] T. Olsson, J. Grundy, Supporting traceability and inconsistency management between software artifacts, in: International Conference on Software Engineering and Applications (SEA 2002), Cambridge, Massachusetts, U.S.A.
[49] P. Fradet, D. Le Metayer, P. Michael, Consistency checking for multiple view software architectures, in: Software Engineering - ESEC/FSE'99, Toulouse, France.
[50] R. Kazman, S. G. Woods, S. J. Carriere, Requirements for integrating software architecture and reengineering models: Corum ii, in: Working Conference on Reverse Engineering (WCRE 1998), Honolulu, Hawaii, U.S.A.
[51] J. L. Cánovas Izquierdo, J. García Molina, An architecture-driven modernization tool for calculating metrics, IEEE Software 27 (2010) 37–43.
[52] Omg object constraint language (ocl), `http://www.omg.org/spec/OCL`, Accessed February 17, 2014.
[53] Kermeta, `http://www.kermeta.org/`, Accessed September 16, 2013.
[54] Eclipse atl project, `http://eclipse.org/atl`, Accessed February 17, 2014.
[55] Eclipse modisco project, `http://www.eclipse.org/MoDisco`, Accessed February 17, 2014.
[56] Atlanmod team (inria, emn and lina), `http://www.emn.fr/x-info/atlanmod`, Accessed February 17, 2014.
[57] Mia-software, `www.mia-software.com`, Accessed February 17, 2014.
[58] Eclipse emf facet project, `http://www.eclipse.org/modeling/emft/facet`, Accessed February 17, 2014.
[59] Eclipse extension points, `http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points\%3F`, Accessed February 17, 2014.
[60] Eclipse public license (epl), `http://www.eclipse.org/legal/epl-v10.html`, Accessed February 17, 2014.
[61] Eclipse cdo project, `http://eclipse.org/cdo`, Accessed February 17, 2014.
[62] Checkstyle development tool, `http://checkstyle.sourceforge.net`, Accessed February 17, 2014.
[63] Sonar tool, `http://www.sonarsource.org`, Accessed February 17, 2014.
[64] Omg xml metadata interchange (xmi), `http://www.omg.org/spec/XMI`, Accessed February 17, 2014.
[65] Eclipse modisco project's wiki, `http://wiki.eclipse.org/MoDisco`, Accessed February 17, 2014.
[66] Eclipse modisco project's official documentation, `http://help.eclipse.org`, Accessed February 17, 2014.
[67] Eclipse development process, `http://www.eclipse.org/projects/dev_process/development_process.php`, Accessed February 17, 2014.
[68] Artist european project (fp7-ict), `http://www.artist-project.eu`, Accessed February 17, 2014.